

345

CD-ROM  
Included



# The AMD-K6 3D Processor

Revolutionary Multimedia Performance

# The AMD-K6 3D Processor

**Revolutionary Multimedia Performance**

**Foreword by  
John C. Dvorak**

This book  
describes the  
AMD-K6<sup>®</sup>-2  
Processor!

AMD  
K  
2  
P R O C E S S O R

Designed for  
Microsoft  
Windows 95

**BONUS!**  
**CD-ROM**  
Features Simulator  
that teaches how  
to program  
your CPU

**ARCUS**

**Edited by Howard Kalish and Jerry Isaac**



#### **Trademarks**

AMD, the AMD logo, and combinations thereof, K86, AMD-K5, the AMD-K6 logo, and Super7 are trademarks, and RISC86 and AMD-K6 are registered trademarks of Advanced Micro Devices, Inc.

Microsoft and Windows are registered trademarks, and Windows NT is a trademark of Microsoft Corporation.

Netware is a registered trademark of Novell, Inc.

MMX is a trademark and Pentium is a registered trademark of the Intel Corporation.

The TAP State Diagram is reprinted from IEEE Std 1149.1-1990 "IEEE Standard Test Access Port and Boundary-Scan Architecture," Copyright © 1990 by the Institute of Electrical and Electronics Engineers, Inc. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner. Information is reprinted with the permission of the IEEE.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

---

**The**  
**AMD-K6<sup>®</sup> 3D**  
**Processor**

**Revolutionary Multimedia  
Performance**

**Edited by Howard Kalish and Jerry Isaac**

Copyright © 1998

Abacus  
5370 52<sup>nd</sup> Street SE  
Grand Rapids, MI 49512  
[www.abacuspublisher.com](http://www.abacuspublisher.com)

This book is copyrighted. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of Abacus Software.

Every effort has been made to ensure complete and accurate information concerning the material presented in this book. However, Abacus Software can neither guarantee nor be held legally responsible for any mistakes in printing or faulty instructions contained in this book. The authors always appreciate receiving notice of any errors or misprints.

This book contains trade names and trademarks of several companies. Any mention of these names or trademarks in this book are not intended to either convey endorsement or other associations with this book. The programs on the CD-ROM are understood to be copyrighted by AMD, Inc.

Printed in the U.S.A.

ISBN 1-55755-345-9

10 9 8 7 6 5 4 3 2 1

# Contents

<b>Killer Chips of the 21st Century</b>	<b>xxv</b>
By John C. Dvorak	
<b>Acknowledgments</b>	<b>xxvii</b>
<b>About This Book and CD-ROM</b>	<b>xxviii</b>
<b>1 AMD-K6™ 3D Processor</b>	<b>1</b>
Super7™ Platform Initiative . . . . .	3
Super7 Enhancements . . . . .	3
Super7 Advantages . . . . .	4
<b>2 Internal Architecture</b>	<b>5</b>
Introduction . . . . .	5
AMD-K6 3D Processor Microarchitecture Overview . . . . .	5
Enhanced RISC86® Microarchitecture . . . . .	6
Cache, Instruction Prefetch, and Predecode Bits . . . . .	10
Instruction Fetch and Decode . . . . .	12
Centralized Scheduler . . . . .	16
Execution Units . . . . .	18
Branch-Prediction Logic . . . . .	21
<b>3 Software Environment</b>	<b>23</b>
Registers . . . . .	23
General-Purpose Registers . . . . .	23
Integer Data Types . . . . .	25
Segment Registers . . . . .	26
Segment Usage . . . . .	26
Instruction Pointer . . . . .	27
Floating-Point Registers . . . . .	27
Floating-Point Register Data Types . . . . .	30
MMX™/3D Registers . . . . .	31
MMX Data Types . . . . .	31
3D Data Types . . . . .	32



EFLAGS Register .....	33
Control Registers .....	34
Debug Registers .....	36
Model-Specific Registers (MSR) .....	39
Memory Management Registers .....	42
Task State Segment .....	44
Paging .....	45
Descriptors and Gates .....	48
Exceptions and Interrupts .....	51
Instructions Supported by the Processor .....	52
<b>4 3D Technology</b>	<b>81</b>
Introduction .....	81
Key Functionality .....	82
3D Feature Detection .....	83
3D Register Set .....	84
3D Data Type Details .....	85
3D Instruction Formats .....	87
3D Definitions .....	88
3D Execution Resources .....	89
Task Switching .....	93
3D Exceptions .....	93
Prefixes .....	94
3D Instruction Coding .....	95
Division and Square Root .....	95
3D Instruction Set .....	95
FEMMS .....	96
PAVGUSB .....	97
PF2ID .....	99
PFACC .....	101
PFADD .....	103
PFCMPEQ .....	105
PFCMPGE .....	107
PFCMPGT .....	109

PFFMAX	111
PFFMIN	113
PFFMUL	115
PFFRCPP	117
PFFRCPPIT1	119
PFFRCPPIT2	121
PFFRSQIT1	123
PFFRSQRT	125
PFFSUB	127
PFFSUBR	129
PI2FD	131
PMULHRW	132
PREFETCH/PREFETCHW	134

## 5 Signal Descriptions 137

Signal Terminology.....	139
A20M# (Address Bit 20 Mask).....	140
A[31:3] (Address Bus).....	141
ADS# (Address Strobe).....	142
ADSC# (Address Strobe Copy) .....	142
AHOLD (Address Hold) .....	143
AP (Address Parity) .....	144
APCHK# (Address Parity Check) .....	145
BE[7:0]# (Byte Enables) .....	146
BF[2:0] (Bus Frequency).....	147
BOFF# (Backoff) .....	148
BRDY# (Burst Ready).....	149
BRDYC# (Burst Ready Copy) .....	150
BREQ (Bus Request).....	151
CACHE# (Cacheable Access).....	152
CLK (Clock) .....	152
D/C# (Data/Code) .....	153

**v**

D[63:0] (Data Bus) . . . . .	154
DP[7:0] (Data Parity) . . . . .	155
EADS# (External Address Strobe) . . . . .	156
EWBE# (External Write Buffer Empty) . . . . .	157
FERR# (Floating-Point Error) . . . . .	158
FLUSH# (Cache Flush) . . . . .	159
HIT# (Inquire Cycle Hit) . . . . .	160
HITM# (Inquire Cycle Hit To Modified Line) . . . . .	160
HLDA (Hold Acknowledge) . . . . .	161
HOLD (Bus Hold Request) . . . . .	162
IGNNE# (Ignore Numeric Exception) . . . . .	163
INIT (Initialization) . . . . .	164
INTR (Maskable Interrupt) . . . . .	165
INV (Invalidation Request) . . . . .	165
KEN# (Cache Enable) . . . . .	166
LOCK# (Bus Lock) . . . . .	167
M/IO# (Memory or I/O) . . . . .	168
NA# (Next Address) . . . . .	169
NMI (Non-Maskable Interrupt) . . . . .	170
PCD (Page Cache Disable) . . . . .	171
PCHK# (Parity Check) . . . . .	172
PWT (Page Writethrough) . . . . .	173
RESET (Reset) . . . . .	174
RSVD (Reserved) . . . . .	175
SCYC (Split Cycle) . . . . .	175
SMI# (System Management Interrupt) . . . . .	176
SMIACT# (System Management Interrupt Active) . . . . .	177
STPCLK# (Stop Clock) . . . . .	178
TCK (Test Clock) . . . . .	179
TDI (Test Data Input) . . . . .	179

TDO (Test Data Output).....	179
TMS (Test Mode Select).....	180
TRST# (Test Reset).....	180
VCC2DET (VCC2 Detect).....	181
VCC2H/L# (VCC2 High/Low).....	181
W/R# (Write/Read).....	182
WB/WT# (Writeback or Writethrough).....	183
<b>6 Bus Cycles</b>	<b>187</b>
Timing Diagrams.....	187
Bus State Machine Diagram.....	189
Idle.....	190
Address.....	190
Data.....	190
Data-NA# Requested.....	190
Pipeline Address.....	191
Pipeline Data.....	191
Transition.....	192
Memory Reads and Writes.....	192
Single-Transfer Memory Read and Write.....	192
Misaligned Single-Transfer Memory Read and Write.....	194
Burst Reads and Pipelined Burst Reads.....	195
Burst Writeback.....	197
I/O Read and Write.....	199
Basic I/O Read and Write.....	199
Misaligned I/O Read and Write.....	200
Inquire and Bus Arbitration Cycles.....	201
Hold and Hold Acknowledge Cycle.....	202
HOLD-Initiated Inquire Hit to Shared or Exclusive Line.....	203
HOLD-Initiated Inquire Hit to Modified Line.....	205
AHOLD-Initiated Inquire Miss.....	207
AHOLD-Initiated Inquire Hit to Shared or Exclusive Line.....	209



AHOLD-Initiated Inquire Hit to Modified Line . . . . .	210
AHOLD Restriction . . . . .	212
Bus Backoff (BOFF#) . . . . .	213
Locked Cycles . . . . .	215
Basic Locked Operation . . . . .	215
Locked Operation with BOFF# Intervention . . . . .	216
Interrupt Acknowledge . . . . .	218
Special Bus Cycles . . . . .	220
Basic Special Bus Cycle . . . . .	220
Shutdown Cycle . . . . .	222
Stop Grant and Stop Clock States . . . . .	223
INIT-Initiated Transition from Protected Mode to Real Mode . . . . .	225
<b>7 Power-on Configuration and Initialization . . . . .</b>	<b>227</b>
Signals Sampled During the Falling Transition of RESET . . . . .	227
FLUSH# . . . . .	227
BF[2:0] . . . . .	228
BRDYC# . . . . .	228
RESET Requirements . . . . .	228
State of Processor After RESET . . . . .	229
Output Signals . . . . .	229
Registers . . . . .	229
State of Processor After INIT . . . . .	232
<b>8 Cache Organization . . . . .</b>	<b>233</b>
MESI States in the Data Cache . . . . .	235
Predecode Bits . . . . .	235
Cache Operation . . . . .	235
Cache-Related Signals . . . . .	238
Cache Disabling . . . . .	238
Cache-Line Fills . . . . .	239
Cache-Line Replacements . . . . .	240
Write Allocate . . . . .	240
Write to a Cacheable Page . . . . .	241
Write to a Sector . . . . .	241

Write Allocate Limit .....	242
Descriptions of the Logic Mechanisms and Conditions ....	244
Prefetching .....	245
Cache States .....	246
Cache Coherency .....	247
Inquire Cycles .....	247
Internal Snooping .....	247
FLUSH# .....	248
WBINVD and INVD .....	248
Cache-Line Replacement .....	248
Cache Snooping .....	250
Writethrough vs. Writeback Coherency States .....	251
A20M# Masking of Cache Accesses .....	251
<b>9 Floating-Point and Multimedia Execution Units</b>	<b>253</b>
Floating-Point Execution Unit .....	253
Handling Floating-Point Exceptions .....	254
External Logic Support of Floating-Point Exceptions ....	254
Multimedia and 3D Execution Units .....	255
Floating-Point and MMX/3D Instruction Compatibility .....	256
Registers .....	256
Exceptions .....	256
FERR# and IGNNE# .....	256
<b>10 System Management Mode (SMM)</b>	<b>257</b>
Overview .....	257
SMM Operating Mode and Default Register Values .....	258
SMM State-Save Area .....	260
SMM Revision Identifier .....	262
SMM Base Address .....	263
Halt Restart Slot .....	264
I/O Trap Dword .....	265
I/O Trap Restart Slot .....	266
Exceptions, Interrupts, and Debug in SMM .....	267

<b>11 Test and Debug</b>	<b>269</b>
Built-In Self-Test (BIST) .....	270
Tri-State Test Mode .....	270
Boundary-Scan Test Access Port (TAP) .....	271
Test Access Port .....	271
TAP Signals .....	272
TAP Registers .....	273
TAP Instructions .....	277
TAP Controller State Machine .....	278
L1 Cache Inhibit .....	282
Purpose .....	282
Debug .....	283
Debug Registers .....	283
Debug Exceptions .....	288
<b>12 Clock Control</b>	<b>291</b>
Halt State .....	292
Enter Halt State .....	292
Exit Halt State .....	293
Stop Grant State .....	293
Enter Stop Grant State .....	293
Exit Stop Grant State .....	294
Stop Grant Inquire State .....	295
Enter Stop Grant Inquire State .....	295
Exit Stop Grant Inquire State .....	295
Stop Clock State .....	295
Enter Stop Clock State .....	295
Exit Stop Clock State .....	296
<b>13 Power and Grounding</b>	<b>299</b>
Power Connections .....	299
Decoupling Recommendations .....	301
Pin Connection Requirements .....	301

<b>14 Electrical Data</b>	<b>303</b>
Introduction .....	303
Operating Ranges .....	303
Absolute Ratings .....	304
DC Characteristics .....	305
Power Dissipation .....	307
<b>15 I/O Buffer Characteristics</b>	<b>309</b>
Introduction .....	309
Selectable Drive Strength .....	310
I/O Buffer Model .....	310
I/O Model Application Note .....	312
I/O Buffer AC and DC Characteristics .....	312
<b>16 Signal Switching Characteristics</b>	<b>313</b>
Introduction .....	313
CLK Switching Characteristics .....	314
Clock Switching Characteristics for 100-MHz Bus Operation .....	314
Clock Switching Characteristics for 66-MHz Bus Operation .....	315
Valid Delay, Float, Setup, and Hold Timings .....	316
Output Delay Timings for 100-MHz Bus Operation .....	316
Input Setup and Hold Timings for 100-MHz Bus Operation .....	318
Output Delay Timings for 66-MHz Bus Operation .....	320
Input Setup and Hold Timings for 66-MHz Bus Operation .....	322
RESET and Test Signal Timing .....	324
<b>17 Thermal Design</b>	<b>331</b>
Package Thermal Specifications .....	331
Heat Dissipation Path .....	334
Measuring Case Temperature .....	334

Layout and Airflow Considerations .....	335
Voltage Regulator .....	335
Airflow Management in a System Design .....	336
<b>18 Pins and Packaging</b>	<b>339</b>
Introduction .....	339
Pin Description Diagrams .....	340
Pin Designations .....	342
Package Specifications .....	343
321-Pin Staggered CPGA Package Specification .....	343
<b>19 Ordering Information</b>	<b>345</b>
Standard AMD-K6 3D Processor Model 8 Products .....	346
<b>Appendix A MMX Multimedia Technology</b>	<b>347</b>
Introduction .....	347
MMX Multimedia Technology Architecture .....	348
Key Functionality .....	348
MMX Register Set .....	350
MMX Data Type Details .....	352
MMX Instructions .....	353
MMX Instruction Formats .....	354
MMX Programming Considerations .....	355
MMX Feature Detection .....	355
Task Switching .....	356
MMX Exceptions .....	358
Mixing MMX and Floating-Point Instructions .....	358
Prefixes .....	359
MMX Instruction Set .....	360
EMMS .....	361
MOVD .....	362
MOVQ .....	363
PACKSSDW .....	364
PACKSSWB .....	366
PACKUSWB .....	369

PADDB.....	372
PADDD .....	374
PADDSB .....	376
PADDSW .....	378
PADDUSB .....	380
PADDUSW .....	382
PADDW .....	384
PAND.....	386
PANDN .....	388
PCMPEQB .....	390
PCMPEQD .....	392
PCMPEQW .....	394
PCMPGTB .....	396
PCMPGTD .....	398
PCMPGTW .....	400
PMADDWD .....	402
PMULHW .....	404
PMULLW.....	406
POR .....	408
PSLLD .....	410
PSLLQ .....	412
PSLLW.....	414
PSRAD.....	416
PSRAW .....	418
PSRLD.....	420
PSRLQ.....	422
PSRLW .....	424
PSUBB .....	426
PSUBD.....	428
PSUBSB.....	430
PSUBSW .....	432
PSUBUSB .....	434
PSUBUSW.....	436
PSUBW .....	438
PUNPCKHBW.....	440
PUNPCKHDQ.....	442

---

PUNPCKHWD .....	444
PUNPCKLBW .....	446
PUNPCKLDQ .....	448
PUNPCKLWD .....	450
PXOR .....	452
<b>Appendix B Code Optimization</b>	<b>455</b>
Introduction .....	455
The AMD-K6 Family of Processors .....	456
The AMD-K6 3D Processor .....	456
Execution Units and Dependency Latencies .....	458
Execution Unit Terminology .....	458
Six-Stage Pipeline .....	459
Register Execution Units .....	460
Load Unit .....	462
Store Unit .....	463
Branch Condition Unit .....	464
Floating-Point Unit .....	464
Latencies and Throughput .....	465
Resource Constraints .....	466
Code Sample Analysis .....	466
Optimization Coding Guidelines .....	472
General x86 Optimization Techniques .....	472
General AMD-K6 3D Processor x86 Coding Optimizations .....	474
AMD-K6 3D Processor Integer x86 Coding Optimizations .....	478
AMD-K6 3D Processor Multimedia Coding Optimizations .....	482
Division .....	497
Square Root and Reciprocal Square Root .....	498
x87 Floating-Point Coding Optimizations .....	500

<b>Appendix C AMD Processor Recognition</b>	<b>505</b>
Introduction . . . . .	505
Using the CPUTD Instruction . . . . .	506
Overview . . . . .	506
Testing for the CPUTD Instruction . . . . .	506
Using CPUTD Functions . . . . .	507
Identifying the Processor's Vendor . . . . .	508
Determining the Processor Signature (Standard Function) . . . . .	509
Identifying Supported Features . . . . .	510
Testing For Extended Functions . . . . .	513
Determining the Processor Signature (Extended Function) . . . . .	513
Displaying the Processor's Name . . . . .	514
Displaying Cache Information . . . . .	514
Sample Code . . . . .	514
CPUTD . . . . .	515
Standard Functions . . . . .	516
Extended Functions . . . . .	518
Values Returned by the CPUTD Instruction . . . . .	523
<b>Index</b>	<b>525</b>



*xvi*

# List of Figures

Figure 1.	AMD-K6 3D Processor Block Diagram .....	7
Figure 2.	Cache Sector Organization .....	11
Figure 3.	The Instruction Buffer .....	12
Figure 4.	Processor Decode Logic .....	13
Figure 5.	Processor Scheduler .....	17
Figure 6.	Register X and Y Functional Units .....	20
Figure 7.	EAX Register with 16-Bit and 8-Bit Name Components .....	24
Figure 8.	Integer Data Registers .....	25
Figure 9.	Segment Register .....	26
Figure 10.	Segment Usage .....	27
Figure 11.	Floating-Point Register .....	28
Figure 12.	FPU Status Word Register .....	28
Figure 13.	FPU Control Word Register .....	29
Figure 14.	FPU Tag Word Register .....	29
Figure 15.	Packed Decimal Data Register .....	30
Figure 16.	Precision Real Data Registers .....	30
Figure 17.	MMX Data Types .....	31
Figure 18.	3D Data Types .....	32
Figure 19.	EFLAGS Registers .....	33
Figure 20.	Control Register 4 (CR4) .....	34
Figure 21.	Control Register 3 (CR3) .....	34
Figure 22.	Control Register 2 (CR2) .....	34
Figure 23.	Control Register 1 (CR1) .....	35
Figure 24.	Control Register 0 (CR0) .....	35
Figure 25.	Debug Register DR7 .....	36
Figure 26.	Debug Register DR6 .....	37
Figure 27.	Debug Registers DR5 and DR4 .....	37
Figure 28.	Debug Registers DR3, DR2, DR1, and DR0 .....	38
Figure 29.	Machine-Check Address Register (MCAR) .....	39
Figure 30.	Machine-Check Type Register (MCTR) .....	40
Figure 31.	Test Register 12 (TR12) .....	40
Figure 32.	Time Stamp Counter (TSC) .....	40

Figure 33.	Extended Feature Enable Register (EFER).....	41
Figure 34.	SYSCALL/SYSRET Target Address Register (STAR) .....	41
Figure 35.	Write Handling Control Register (WHCR).....	42
Figure 36.	Memory Management Registers .....	43
Figure 37.	Task State Segment (TSS) .....	44
Figure 38.	4-Kbyte Paging Mechanism .....	45
Figure 39.	4-Mbyte Paging Mechanism .....	46
Figure 40.	Page Directory Entry 4-Kbyte Page Table (PDE) ....	47
Figure 41.	Page Directory Entry 4-Mbyte Page Table (PDE) ....	47
Figure 42.	Page Table Entry (PTE) .....	48
Figure 43.	Application Segment Descriptor .....	49
Figure 44.	System Segment Descriptor .....	50
Figure 45.	Gate Descriptor .....	51
Figure 46.	3D/MMX Registers .....	84
Figure 47.	3D Data Type Details .....	85
Figure 48.	Single-Precision, Floating-Point Data Format .....	86
Figure 49.	Integer Data Types .....	86
Figure 50.	Register X Unit and Register Y Unit Resources .....	91
Figure 51.	Logic Symbol Diagram .....	138
Figure 52.	Waveform Definitions .....	188
Figure 53.	Bus State Machine Diagram .....	189
Figure 54.	Non-Pipelined Single-Transfer Memory Read/Write and Write Delayed by EWBE# .....	193
Figure 55.	Misaligned Single-Transfer Memory Read and Write .....	195
Figure 56.	Burst Reads and Pipelined Burst Reads .....	197
Figure 57.	Burst Writeback due to Cache-Line Replacement .....	198
Figure 58.	Basic I/O Read and Write .....	200
Figure 59.	Misaligned I/O Transfer .....	201
Figure 60.	Basic HOLD/HLDA Operation .....	203
Figure 61.	HOLD-Initiated Inquire Hit to Shared or Exclusive Line .....	204
Figure 62.	HOLD-Initiated Inquire Hit to Modified Line .....	206
Figure 63.	AHOLD-Initiated Inquire Miss .....	208

Figure 64.	AHOLD-Initiated Inquire Hit to Shared or Exclusive Line.....	209
Figure 65.	AHOLD-Initiated Inquire Hit to Modified Line.....	211
Figure 66.	AHOLD Restriction.....	213
Figure 67.	BOFF# Timing .....	214
Figure 68.	Basic Locked Operation .....	216
Figure 69.	Locked Operation with BOFF# Intervention .....	217
Figure 70.	Interrupt Acknowledge Operation .....	219
Figure 71.	Basic Special Bus Cycle (Halt Cycle).....	221
Figure 72.	Shutdown Cycle.....	222
Figure 73.	Stop Grant and Stop Clock Modes, Part 1.....	224
Figure 74.	Stop Grant and Stop Clock Modes, Part 2.....	225
Figure 75.	INIT-Initiated Transition from Protected Mode to Real Mode .....	226
Figure 76.	Cache Organization.....	234
Figure 77.	Cache Sector Organization.....	234
Figure 78.	Write Handling Control Register (WHCR).....	242
Figure 79.	Write Allocate Logic Mechanisms and Conditions .....	243
Figure 80.	External Logic for Supporting Floating-Point Exceptions .....	255
Figure 81.	SMM Memory.....	259
Figure 82.	TAP State Diagram .....	279
Figure 83.	Debug Register DR7 .....	284
Figure 84.	Debug Register DR6 .....	285
Figure 85.	Debug Registers DR5 and DR4 .....	285
Figure 86.	Debug Registers DR3, DR2, DR1, and DR0 .....	286
Figure 87.	Clock Control State Transitions.....	297
Figure 88.	Suggested Component Placement .....	300
Figure 89.	K6STD Pulldown V/I Curves .....	311
Figure 90.	K6STD Pullup V/I Curves.....	311
Figure 91.	CLK Waveform .....	315
Figure 92.	Diagrams Key.....	327
Figure 93.	Output Valid Delay Timing .....	327
Figure 94.	Maximum Float Delay Timing .....	328
Figure 95.	Input Setup and Hold Timing.....	328
Figure 96.	Reset and Configuration Timing .....	329

Figure 97.	TCK Waveform .....	330
Figure 98.	TRST# Timing .....	330
Figure 99.	Test Signal Timing Diagram.....	330
Figure 100.	Thermal Model.....	332
Figure 101.	Power Consumption vs. Thermal Resistance .....	333
Figure 102.	Processor Heat Dissipation Path .....	334
Figure 103.	Measuring Case Temperature .....	334
Figure 104.	Voltage Regulator Placement .....	335
Figure 105.	Airflow for a Heatsink with Fan .....	336
Figure 106.	Airflow Path in a Dual-fan System .....	336
Figure 107.	Airflow Path in an ATX Form-Factor System.....	337
Figure 108.	Processor Top-Side View .....	340
Figure 109.	Processor Pin-Side View.....	341
Figure 110.	321-Pin Staggered CPGA Package Specification ....	344
Figure 111.	MMX Registers .....	351
Figure 112.	MMX Data Types.....	353
Figure 113.	Cooperative Task Switching.....	357
Figure 114.	Preemptive Task Switching .....	358
Figure 115.	Processor Pipeline.....	459
Figure 116.	Register X and Y Execution Stages.....	461
Figure 117.	Load Execution Unit.....	462
Figure 118.	Store Unit Execution Pipeline.....	464
Figure 119.	Contents of EAX Register Returned by Function 1.....	509
Figure 120.	Contents of EAX Register Returned by Extended Function 8000_0001h.....	513

# List of Tables

Table 1.	Execution Latency and Throughput of Execution Units . . . . .	19
Table 2.	General-Purpose Registers . . . . .	24
Table 3.	General-Purpose Register Dword, Word, and Byte Names . . . . .	25
Table 4.	Segment Registers . . . . .	26
Table 5.	Model-Specific Registers (MSRs) . . . . .	39
Table 6.	Extended Feature Enable Register (EFER) Definition . . . . .	41
Table 7.	SYSCALL/SYSRET Target Address Register (STAR) Definition . . . . .	42
Table 8.	Memory Management Registers . . . . .	42
Table 9.	Application Segment Types . . . . .	49
Table 10.	System Segment and Gate Types . . . . .	50
Table 11.	Summary of Exceptions and Interrupts . . . . .	51
Table 12.	Integer Instructions . . . . .	53
Table 13.	Floating-Point Instructions . . . . .	71
Table 14.	MMX Instructions . . . . .	75
Table 15.	3D Instructions . . . . .	79
Table 16.	3D Technology Exponent Ranges . . . . .	89
Table 17.	3D Floating-Point Instructions . . . . .	92
Table 18.	3D Performance-Enhancement Instructions . . . . .	92
Table 19.	3D and MMX Instruction Exceptions . . . . .	93
Table 20.	Numerical Range for the PF2ID Instruction . . . . .	100
Table 21.	Numerical Range for the PFACC Instruction . . . . .	102
Table 22.	Numerical Range for the PFADD Instruction . . . . .	104
Table 23.	Numerical Range for the PFCMPEQ Instruction . . . . .	106
Table 24.	Numerical Range for the PFCMPGE Instruction . . . . .	108
Table 25.	Numerical Range for the PFCMPGT Instruction . . . . .	110
Table 26.	Numerical Range for the PFMAX Instruction . . . . .	112
Table 27.	Numerical Range for the PFMIN Instruction . . . . .	114
Table 28.	Numerical Range for the PFMUL Instruction . . . . .	116
Table 29.	Numerical Range for the PFRCP Instruction . . . . .	118

Table 30.	Numerical Range for the PFRCPIT1 Instruction . . . . .	120
Table 31.	Numerical Range for the PFRCPIT2 Instruction . . . . .	122
Table 32.	Numerical Range for the PFSQIT1 Instruction. . . . .	124
Table 33.	Numerical Range for PFRSQRT Instruction . . . . .	126
Table 34.	Numerical Range for the PFSUB Instruction . . . . .	128
Table 35.	Numerical Range for the PFSUBR Instruction . . . . .	130
Table 36.	Summary of PREFETCH Instruction Type Options . . . . .	135
Table 37.	Processor-to-Bus Clock Ratios. . . . .	147
Table 38.	Output Pin Float Conditions . . . . .	181
Table 39.	Input Pin Types. . . . .	184
Table 40.	Output Pin Float Conditions . . . . .	185
Table 41.	Input/Output Pin Float Conditions. . . . .	185
Table 42.	Test Pins . . . . .	185
Table 43.	Bus Cycle Definition . . . . .	186
Table 44.	Special Cycles. . . . .	186
Table 45.	Bus-Cycle Order During Misaligned Transfers . . . . .	194
Table 46.	A[4:3] Address-Generation Sequence During Bursts. . . . .	196
Table 47.	Bus-Cycle Order During Misaligned I/O Transfers . . . . .	200
Table 48.	Interrupt Acknowledge Operation Definition. . . . .	218
Table 49.	Encodings For Special Bus Cycles . . . . .	220
Table 50.	Output Signal State After RESET . . . . .	229
Table 51.	Register State After RESET . . . . .	230
Table 52.	PWT Signal Generation . . . . .	237
Table 53.	PCD Signal Generation . . . . .	237
Table 54.	CACHE# Signal Generation . . . . .	237
Table 55.	Data Cache States for Read and Write Accesses . . . . .	246
Table 56.	Cache States for Inquiries, Snoops, Invalidation, and Replacement. . . . .	249
Table 57.	Snoop Action. . . . .	250
Table 58.	Initial State of Registers in SMM. . . . .	259
Table 59.	SMM State-Save Area Map . . . . .	260
Table 60.	SMM Revision Identifier . . . . .	263
Table 61.	I/O Trap Dword Configuration. . . . .	265
Table 62.	I/O Trap Restart Slot . . . . .	266
Table 63.	Boundary Scan Bit Definitions . . . . .	275

Table 64.	Device Identification Register .....	276
Table 65.	Supported Tap Instructions.....	277
Table 66.	DR7 LEN and RW Definitions.....	288
Table 67.	Operating Ranges.....	303
Table 68.	Absolute Ratings .....	304
Table 69.	DC Characteristics .....	305
Table 70.	Typical and Maximum Power Dissipation .....	307
Table 71.	A[20:3], ADS#, HITM#, and W/R# Strength Selection .....	310
Table 72.	CLK Switching Characteristics for 100-MHz Bus Operation .....	314
Table 73.	CLK Switching Characteristics for 66-MHz Bus Operation .....	315
Table 74.	Output Delay Timings for 100-MHz Bus Operation .....	316
Table 75.	Input Setup and Hold Timings for 100-MHz Bus Operation .....	318
Table 76.	Output Delay Timings for 66-MHz Bus Operation ....	320
Table 77.	Input Setup and Hold Timings for 66-MHz Bus Operation .....	322
Table 78.	RESET and Configuration Signals for 100-MHz Bus Operation .....	324
Table 79.	RESET and Configuration Signals for 66-MHz Bus Operation .....	325
Table 80.	TCK Waveform and TRST# Timing at 25 MHz .....	326
Table 81.	Test Signal Timing at 25 MHz .....	326
Table 82.	Package Thermal Specification .....	331
Table 83.	Processor Functional Grouping.....	342
Table 84.	321-Pin Staggered CPGA Package Specification .....	343
Table 85.	Valid Ordering Part Number Combinations .....	346
Table 86.	RISC86 Execution Latencies and Throughput .....	465
Table 87.	Sample 1 – Integer Register Operations .....	468
Table 88.	Sample 2 – Integer Register and Memory Load Operations.....	469
Table 89.	Sample 3 – Integer Register and Memory Load/Store Operations.....	470



Table 90.	Sample 4 – Integer, MMX, and Memory Load/Store Operations.....	471
Table 91.	Decode Accumulation and Serialization .....	475
Table 92.	Summary of CPUID Functions in AMD Processors .....	508
Table 93.	Summary of Processor Signatures for AMD Processors .....	510
Table 94.	Summary of Standard Feature Bits for AMD Processors .....	511
Table 95.	Summary of Extended Feature Bits for AMD Processors .....	512
Table 96.	Standard Feature Flag Descriptions .....	517
Table 97.	Extended Feature Flag Descriptions .....	519
Table 98.	EBX Format Returned by Function 8000_0005h.....	521
Table 99.	ECX Format Returned by Function 8000_0005h.....	521
Table 100.	EDX Format Returned by Function 8000_0005h .....	521
Table 101.	ECX Format Returned by Function 8000_0006h.....	522
Table 102.	Values Returned By AMD Processors .....	523

# Killer Chips of the 21st Century

## FOREWORD

**John C. Dvorak**

The AMD-K6® 3D processor is a miraculous chip. Not only because it incorporates a new design technology and in many ways modernizes the way chips will be designed in the future, but because it reestablishes a key second-source for microprocessors. This is critical to the growth of the industry.

This excellent book and the associated CD-ROM tell you everything you ever wanted to know about the new AMD-K6 3D processor, and then some.

The AMD-K6 3D processor operates at internal clock speeds of 300 MHz and above with a 100-MHz bus. The version of the AMD-K6 3D that is due out before the end of 1998 will operate at 400 MHz with a 100-MHz bus *and* a backside on-chip full-speed L2 cache that runs at the same frequency as the internal processor clock. All of this tremendous performance takes advantage of the cost-effective Socket 7 and Super7™ platforms. This stretches the useful life of this technology and saves everyone money in the process. Add the fact that the AMD-K6 3D 100-MHz motherboards are implementing AGP technology and you have a real-world, cost-effective Pentium® II competitor.

And then there's AMD's 3D technology. This puppy provides a huge improvement in video and multimedia performance. Once again AMD is leading the way in developing and producing an important enhancement to the x86 architecture.

I'm convinced that if it wasn't for AMD and others, you'd be paying twice as much for basic Pentium MMX™ processors, and the Pentium II would probably just be entering the marketplace at astronomical price levels. Competition is good for the consumer. The competition that AMD provides in the PC market makes Intel stay on their toes instead of resting on their laurels.

**XXV**

---

The 3D and multimedia markets are in the midst of rapid growth. Last year's MMX technology started to address this, but most see it as too little and too late. Fact is, there is plenty of room for improvement. MMX doesn't address floating-point computations, the heart and soul of 3D graphics and advanced multimedia applications. The AMD-K6 3D processor with 3D technology confronts this need by implementing a comprehensive set of advanced 3D instructions focused primarily on floating-point operations. These new instructions were developed based on input from the leading game developers, who told AMD exactly what they needed to make their software significantly faster. AMD listened to them and they have implemented a real improvement to the x86 instruction set. You will be able to see this improvement in the performance of the AMD-K6 3D processor. The improvement is not trivial.

A single AMD-K6 3D instruction can perform two 32-bit floating-point operations in one processor clock cycle. And the AMD-K6 3D processor has two pipelines for 3D instructions, which means that the processor can execute two 3D instructions per clock for throughput of up to *four* floating-point operations per clock cycle. That equates to 1.2 GigaFlops on an AMD-K6 3D/300 processor compared to only 0.3 GigaFlop capability on a Pentium II 300. By the way, you can actually see the internal operation of the AMD-K6 3D processor with the simulator that is included on the CD-ROM that comes with this book. If you ever wondered how a RISC processor manages to execute x86 CISC code, the simulator will show you some of the under-the-hood techniques that make it all possible. Fascinating.

And things won't stop with the AMD-K6 3D processor. AMD is not a one-trick pony. The AMD-K7™ is expected to have first silicon soon and be in production in 1999. AMD will continue to provide viable competition to Intel well into the 21st century, and that competition will insure that you will be able to buy leading-edge technology at the lowest possible price. As far as I'm concerned this is the kind of competition that keeps things interesting. This will be a fun battle to watch and in the meantime we all benefit from these great new chips.

John C. Dvorak

Berkeley, California 1998

# ***Acknowledgments***

Many people contributed to this book, including the following major contributors. Our special thanks to everyone who had a part in making this book possible and also to the team of designers and engineers who made the personal sacrifices that were required to make the dream of the AMD-K6 3D processor a reality.

## ***Primary Contributors***

---

Larry Barnett  
Dervinn Caldwell  
Kam Chow  
Greg Favor  
Jeff Ferris  
Kevin Krewell  
Dina Lloreda  
Randy Martin  
Jim Reilly  
Lance Smith  
Dan Wax

# ***About This Book and CD-ROM***

## **The Book**

At the time of publication, AMD had not made final naming decisions for the processor and the 3D technology. The names used in this book are the AMD code names for the processor and the 3D technology.

Refer to Appendix B, “Code Optimization” on page 455 for details regarding the examples shown in the AMD-K6 3D simulator, especially the tables beginning with Table 87 on page 468.

Refer to the AMD web site at [www.amd.com](http://www.amd.com) for updates to material related to this book, including new scripts and updates for the AMD-K6 3D simulator.

## **CD-ROM Contents**

The CD-ROM included with this book contains the following:

- AMD-K6 3D processor simulator
- All AMD processor technical documentation in Adobe Acrobat PDF format
- Adobe Acrobat Reader for most platforms

## **Minimum System Requirements**

The following minimum system is required in order to run the AMD-K6 3D processor simulator:

- A 133-MHz AMD-K6 processor
- 16 Mbytes of memory
- Windows® 95 or Windows NT™ 4.0 operating system
- 256-color SVGA graphics mode video
- Video resolution of 800 by 600

We recommend the following system for maximum enjoyment when using the simulator:

- A 166-MHz AMD-K6 processor or better
- 32 Mbytes of memory
- Windows 95 or Windows NT 4.0 operating system
- 65,536-color graphics mode or better
- Video resolution of 1024 by 768
- A sound card with speakers

## **Installation of the CD-ROM**

A setup program is provided for your convenience. Run `install.exe` from the root directory of the CD-ROM, and the installation program will step you through installing the simulator and the Acrobat reader, if you need it.

***xxviii***

## AMD-K6 3D Processor

- Advanced 6-Issue RISC86® Superscalar Microarchitecture
  - ◆ Ten parallel specialized execution units
  - ◆ Multiple sophisticated x86-to-RISC86 instruction decoders
  - ◆ Advanced two-level branch prediction
  - ◆ Speculative execution
  - ◆ Out-of-order execution
  - ◆ Register renaming and data forwarding
  - ◆ Issues up to six RISC86 instructions per clock
- Large On-Chip Split 64-Kbyte Level-One (L1) Cache
  - ◆ 32-Kbyte instruction cache with additional 20-Kbytes of predecode cache
  - ◆ 32-Kbyte writeback dual-ported data cache
  - ◆ Two-way set associative
  - ◆ MESI protocol support
- 3D Technology
  - ◆ Additional instructions to improve 3D graphics and multimedia performance
  - ◆ Separate multiplier and ALU for superscalar instruction execution
- Compatible with both Super7™ and Socket 7
  - ◆ Compatible with both the 66-MHz processor bus and 100-MHz processor bus
  - ◆ Accelerated Graphic Port (AGP) support
- High-Performance IEEE 754-Compatible and 854-Compatible Floating-Point Unit
- High-Performance Industry-Standard MMX™ Instructions
  - ◆ Dual integer ALU for superscalar execution
- 321-Pin Ceramic Pin Grid Array (CPGA) Package
- Industry-Standard System Management Mode (SMM)
- IEEE 1149.1 Boundary Scan
- Full x86 Binary Software Compatibility

# **1** *AMD-K6 3D Processor*

As the newest member of the AMD K86™ family of x86 processors, the innovative AMD-K6 3D processor brings industry-leading performance to PC systems running the extensive installed base of x86 software. Its Super7-compatible, 321-pin ceramic pin grid array (CPGA) package enables the processor to reduce time-to-market by leveraging today's cost-effective infrastructure to deliver a superior price/performance PC solution.

The AMD-K6 3D processor is the first to incorporate 3D technology, a significant innovation to the x86 processor architecture that drives today's personal computers. With 3D technology, new, more powerful hardware and software applications enable a more entertaining and productive PC platform. Improvements include faster frame rates on high-resolution scenes, superior modeling of real world environments and physics, sharper and more detailed 3D imaging, smoother video playback, and near theater-quality audio.

AMD has taken a leadership role in developing new instructions that enable exciting new levels of performance and realism. 3D technology was defined and implemented in collaboration with Microsoft®, application developers, and graphics vendors, and has received an enthusiastic reception. It is compatible with today's existing x86 software and requires no operating system support, thereby enabling 3D applications to work with all existing operating systems.

To provide state-of-the-art performance, the processor incorporates the innovative and efficient RISC86 microarchitecture, a large 64-Kbyte level-one cache (32-Kbyte dual-ported data cache, 32-Kbyte instruction cache with an additional 20-Kbytes of predecode cache), a powerful IEEE 754-compatible and 854-compatible floating-point execution unit, and a high-performance industry-standard multimedia execution unit for executing MMX instructions. The processor includes additional high-performance Single Instruction Multiple Data (SIMD) execution resources to support the 3D technology. These techniques have been combined to deliver industry leadership in 16-bit and 32-bit performance, providing exceptional performance for both Windows 95 and Windows NT software bases.

The AMD-K6 3D processor's 6-issue RISC86 microarchitecture is a decoupled decode/execution superscalar design that implements state-of-the-art design techniques to achieve leading-edge performance. Advanced design techniques implemented in the AMD-K6 3D processor include multiple x86 instruction decode, single-clock internal RISC operations, ten execution units that support superscalar operation, out-of-order execution, data forwarding, speculative execution, and register renaming. In addition, the processor supports the industry's most advanced branch prediction logic by implementing an 8192-entry branch history table, the industry's only branch target cache, and a return address stack, which combine to deliver better than a 95% prediction rate. These design techniques enable the AMD-K6 3D processor to issue, execute, and retire multiple x86 instructions per clock, resulting in excellent scaleable performance.

## **2**

The AMD-K6 3D processor is fully x86 binary code compatible. AMD's extensive experience through six generations of x86 processors has been carefully integrated into the processor to ensure complete compatibility with Windows 9x, Windows 3.x, Windows NT, DOS, OS/2, Unix, Solaris, NetWare®, Vines, and other leading x86 operating systems and applications. The AMD-K6 3D processor is Super7 and Socket 7-compatible. The Super7 initiative is an extension to today's popular and robust Socket 7 platform. See "Super7 Platform Initiative" for more information.

AMD has designed, manufactured, and delivered over 50 million Microsoft Windows-compatible processors in the last five years alone. The AMD-K6 3D processor is the latest member in this long line of processors. With its combination of state-of-the-art features, industry-leading performance, high-performance 3D and multimedia engines, full x86 compatibility, and low-cost infrastructure, the AMD-K6 3D is the superior choice for mainstream personal computers.

## Super7 Platform Initiative

---

AMD and its industry partners are investing in the future of Socket 7 with the new Super7 platform initiative. The goal of the initiative is to maintain the competitive vitality of the Socket 7 infrastructure through a series of planned enhancements, including the development of an industry-standard 100-MHz processor bus protocol.

In addition to the 100-MHz processor bus protocol, the Super7 initiative includes the introduction of chipsets that support the AGP specification, and support for a backside L2 cache and frontside L3 cache.

## Super7 Enhancements

The Super7 platform has the following enhancements:

- *100-MHz processor bus*—The AMD-K6 3D processor supports a 100-MHz, 800 Mbyte/second frontside bus to provide a high-speed interface to Super7 platform-based chipsets. The 100-MHz interface to the frontside Level 2 (L2) cache and main system memory speeds up access to the frontside cache and main memory by 50 percent over the 66-MHz Socket 7 interface—a significant increase in system performance equivalent to a jump of up to two processor speed grades.
- *Accelerated graphics port support*—AGP improves the performance of mid-range PCs that have small amounts of video memory on the graphics card. The industry-standard AGP specification enables a 133-MHz graphics interface and will scale to even higher levels of performance.
- *Support for backside L2 and frontside L3 cache*—The Super7 platform has the 'headroom' to support higher-performance AMD-K6 processors, with clock speeds scaling to 400 MHz and beyond. Future versions of the AMD-K6 processor will



# **1** *AMD-K6 3D Processor*

---

feature a full-speed, on-chip backside 256-Kbyte L2 cache designed to deliver new levels of system performance to mainstream desktop systems. These versions of the processor will also support an optional 100-MHz frontside L3 cache for even higher-performance system configurations.

## **Super7 Advantages**

The Super7 platform has the following advantages:

- Delivers performance and features competitive with alternate platforms at the same clock speed, and at a significantly lower cost
- Takes advantage of existing system designs for superior value
- Enables OEMs and resellers to take advantage of mature, high-volume infrastructure supported by multiple BIOS, chipset, graphics, and motherboard suppliers
- Reduces inventory and design costs with one motherboard for a wide range of products
- Builds on a huge installed base of more than 100 million motherboards
- Provides an easy upgrade path for future PC users, as well as a bridge to legacy users

By taking advantage of the low-cost, mature Socket 7 infrastructure, the Super7 platform will continue to provide superior value and leading-edge performance for mainstream desktop systems.

# 2

## Internal Architecture

### Introduction

---

The AMD-K6 3D processor implements advanced design techniques known as the RISC86 microarchitecture. The RISC86 microarchitecture is a decoupled decode/execution design approach that yields superior sixth-generation performance for x86-based software. This chapter describes the techniques used and the functional elements of the RISC86 microarchitecture.

### AMD-K6 3D Processor Microarchitecture Overview

---

When discussing processor design, it is important to understand the terms *architecture*, *microarchitecture*, and *design implementation*. The term *architecture* refers to the instruction set and features that are visible to software programs running on the processor. The architecture determines what software the processor can run. The architecture of the AMD-K6 3D processor is the industry-standard x86 instruction set.

The term *microarchitecture* refers to the design techniques used in the processor to reach the target cost, performance, and functionality goals. The AMD-K6 family of processors are based on a sophisticated RISC core known as the enhanced RISC86 microarchitecture. The enhanced RISC86 microarchitecture is

## 2 Internal Architecture

an advanced, second-order decoupled decode/execution design approach that enables industry-leading performance for x86-based software.

The term *design implementation* refers to the actual logic and circuit designs from which the processor is created according to the microarchitecture specifications.

### Enhanced RISC86 Microarchitecture

The enhanced RISC86 microarchitecture defines the characteristics of the AMD-K6 family. The innovative RISC86 microarchitecture approach implements the x86 instruction set by internally translating x86 instructions into RISC86 operations. These RISC86 operations were specially designed to include direct support for the x86 instruction set while observing the RISC performance principles of fixed-length encoding, regularized instruction fields, and a large register set. The enhanced RISC86 microarchitecture used in the AMD-K6 3D processor enables higher processor core performance and promotes straightforward extensions, such as those added in the current AMD-K6 3D processor and those planned for the future. Instead of directly executing complex x86 instructions, which have lengths of 1 to 15 bytes, the AMD-K6 3D processor executes the simpler and easier fixed-length RISC86 opcodes, while maintaining the instruction coding efficiencies found in x86 programs.

The processor contains parallel decoders, a centralized RISC86 operation scheduler, and ten execution units that support superscalar operation—multiple decode, execution, and retirement—of x86 instructions. These elements are packed into an aggressive and highly efficient six-stage pipeline.

#### Processor Block Diagram

As shown in Figure 1 on page 7, the high-performance, out-of-order execution engine of the AMD-K6 3D processor is mated to a split level-one 64-Kbyte writeback cache with 32 Kbytes of instruction cache and 32 Kbytes of data cache. The instruction cache feeds the decoders and, in turn, the decoders feed the scheduler. The ICU issues and retires RISC86 operations contained in the scheduler. The system bus interface is an industry-standard 64-bit Super7 and Socket 7 demultiplexed bus.

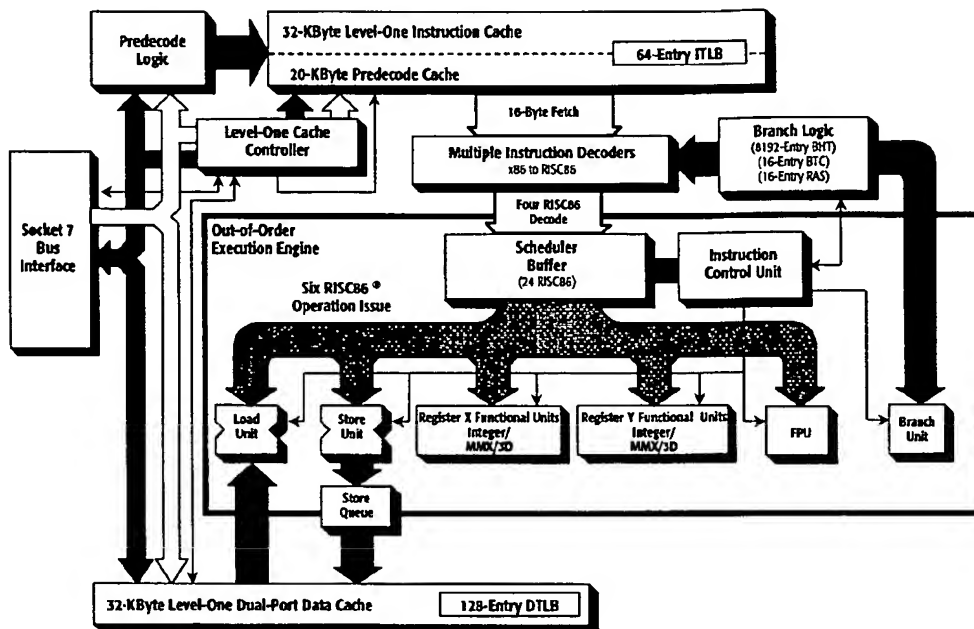


Figure 1. AMD-K6 3D Processor Block Diagram

The AMD-K6 3D processor combines the latest in processor microarchitecture to provide the highest x86 performance for today's personal computers. The processor offers true sixth-generation performance and full x86 binary software compatibility.

#### Decoders

Decoding of the x86 instructions begins when the on-chip instruction cache is filled. Predecode logic determines the length of an x86 instruction on a byte-by-byte basis. This predecode information is stored, along with the x86 instructions, in the instruction cache, to be used later by the decoders. The decoders translate on-the-fly, with no additional latency, up to two x86 instructions per clock into RISC86 operations.

*Note: In this chapter, "clock" refers to a processor clock.*

The AMD-K6 3D processor categorizes x86 instructions into three types of decodes—short, long, and vector. The decoders process either two short, one long, or one vector decode at a time.

## 2 Internal Architecture

---

The three types of decodes have the following characteristics:

- *Short decodes*—x86 instructions less than or equal to seven bytes in length
- *Long decodes*—x86 instructions less than or equal to 11 bytes in length
- *Vector decodes*—complex x86 instructions

Short and long decodes are processed completely within the decoders. Vector decodes are started by the decoders and then completed by fetched sequences from an on-chip ROM. After decoding, the RISC86 operations are delivered to the scheduler for dispatching to the executions units.

### **Scheduler/Instruction Control Unit**

The centralized scheduler or buffer is managed by the Instruction Control Unit (ICU). The ICU buffers and manages up to 24 RISC86 operations at a time. This equals from 6 to 12 x86 instructions. This buffer size (24) is perfectly matched to the processor's six-stage RISC86 pipeline and 4 RISC86-operations decode rate. The scheduler accepts as many as four RISC86 operations at a time from the decoders and retires up to 4 RISC86 operations per clock cycle. The ICU is capable of simultaneously issuing up to six RISC86 operations at a time to the execution units. This consists of the following types of operations:

- Memory load operation
- Memory store operation
- Complex integer, MMX or 3D register operation
- Simple integer, MMX or 3D register operation
- Floating-point register operation
- Branch condition evaluation

### **Registers**

The scheduler uses 48 physical registers that are contained within the RISC86 microarchitecture when managing the 24 RISC86 operations. The 48 physical registers are located in a general register file and are grouped as 24 general registers, plus 24 renaming registers. The 24 general registers consist of 16 scratch registers and eight registers that correspond to the x86 general purpose registers—EAX, EBX, ECX, EDX, EBP, ESP, ESI and EDI.

**Branch Logic**

The AMD-K6 3D processor is designed with highly sophisticated dynamic branch logic consisting of the following:

- Branch history/Prediction table
- Branch target cache
- Return address stack

The processor implements a two-level branch prediction scheme based on an 8192-entry branch history table. The branch history table stores prediction information that is used for predicting conditional branches. Because the branch history table does not store predicted target addresses, special address ALUs calculate target addresses on-the-fly during instruction decode. The branch target cache augments predicted branch performance by avoiding a one clock cache-fetch penalty. This specialized target cache does this by supplying the first 16 bytes of target instructions to the decoders when branches are predicted. The return address stack is a unique device specifically designed for optimizing CALL and RETURN pairs. In summary, the AMD-K6 3D processor uses dynamic branch logic to minimize delays due to the branch instructions that are common in x86 software.

**3D Technology**

AMD has taken a lead role in improving the multimedia and 3D capabilities of the x86 processor family with the introduction of 3D technology, which uses a packed, single-precision, floating-point data format and Single Instruction Multiple Data (SIMD) operations based on the MMX model. For more information, see Chapter 4, “3D Technology” on page 81.

## **2** *Internal Architecture*

---

### **Cache, Instruction Prefetch, and Predecode Bits**

The writeback level-one cache on the AMD-K6 3D processor is organized as a separate 32-Kbyte instruction cache and a 32-Kbyte data cache with two-way set associativity. The cache line size is 32 bytes and lines are prefetched from main memory using an efficient pipelined burst transaction. As the instruction cache is filled, each instruction byte is analyzed for instruction boundaries using predecoding logic. Predecoding annotates information (5 bits per byte) to each instruction byte that later enables the decoders to efficiently decode multiple instructions simultaneously.

#### **Cache**

The processor cache design takes advantage of a sectored organization (see Figure 2 on page 11). Each sector consists of 64 bytes configured as two 32-byte cache lines. The two cache lines of a sector share a common tag but have separate pairs of MESI (Modified, Exclusive, Shared, Invalid) bits that track the state of each cache line.

Two forms of cache misses and associated cache fills can take place—a sector replacement and a cache line replacement. In the case of a sector replacement, the miss is due to a tag mismatch, in which case the required cache line is filled from external memory, and the cache line within the sector that was not required is marked as invalid. In the case of a cache line replacement, the address matches the tag, but the requested cache line is marked as invalid. The required cache line is filled from external memory, and the cache line within the sector that is not required remains in the same cache state.

#### **Prefetching**

The processor performs cache prefetching for sector replacements only—as opposed to cache line replacements. This cache prefetching results in the filling of the required cache line first, and a prefetch of the second cache line. Furthermore, the prefetch of the cache line that is not required is initiated only in the forward direction—that is, only if the requested cache line is the first cache line within the sector. From the perspective of the external bus, the two cache-line fills typically appear as two 32-byte burst read cycles occurring back-to-back or, if allowed, as pipelined cycles.

The 3D technology includes a new instruction called PREFETCH that allows a cache line to be prefetched into the data cache. The PREFETCH instruction format is defined in Table 15, “3D Instructions,” on page 79. For more detailed information, see Chapter 4, “3D Technology” on page 81.

**Predecode Bits**

Decoding x86 instructions is particularly difficult because the instructions are variable-length and can be from 1 to 15 bytes long. Predecode logic supplies the five predecode bits that are associated with each instruction byte. The predecode bits indicate the number of bytes to the start of the next x86 instruction. The predecode bits are stored in an extended instruction cache alongside each x86 instruction byte as shown in Figure 2. The predecode bits are passed with the instruction bytes to the decoders where they assist with parallel x86 instruction decoding.

Tag	Cache Line 1	Byte 31	Predecode Bits	Byte 30	Predecode Bits	.....	.....	Byte 0	Predecode Bits	MESI Bits
Address	Cache Line 2	Byte 31	Predecode Bits	Byte 30	Predecode Bits	.....	.....	Byte 0	Predecode Bits	MESI Bits

**Figure 2. Cache Sector Organization**



## 2 Internal Architecture

### Instruction Fetch and Decode

#### Instruction Fetch

The processor can fetch up to 16 bytes per clock out of the instruction cache or branch target cache. The fetched information is placed into a 16-byte instruction buffer that feeds directly into the decoders (see Figure 3). Fetching can occur along a single execution stream with up to seven outstanding branches taken.

The instruction fetch logic is capable of retrieving any 16 contiguous bytes of information within a 32-byte boundary. There is no additional penalty when the 16 bytes of instructions cross a cache-line boundary. The instruction bytes are loaded into the instruction buffer as they are consumed by the decoders. Although instructions can be consumed with byte granularity, the instruction buffer is managed on a memory-aligned word (two bytes) organization. Therefore, instructions are loaded and replaced with word granularity. When a control transfer occurs—such as a JMP instruction—the entire instruction buffer is flushed and reloaded with a new set of 16 instruction bytes.

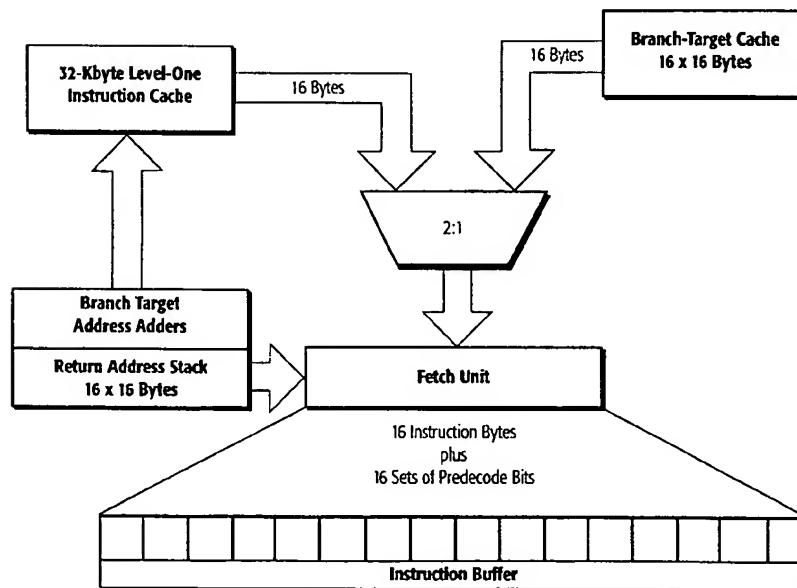


Figure 3. The Instruction Buffer

## Instruction Decode

The AMD-K6 3D processor decode logic is designed to decode multiple x86 instructions per clock (see Figure 4). The decode logic accepts x86 instruction bytes and their predecode bits from the instruction buffer, locates the actual instruction boundaries, and generates RISC86 operations from these x86 instructions.

RISC86 operations are fixed-length internal instructions. Most RISC86 operations execute in a single clock. RISC86 operations are combined to perform every function of the x86 instruction set. Some x86 instructions are decoded into as few as zero RISC86 opcodes—for instance a NOP—or one RISC86 operation—a register-to-register add. More complex x86 instructions are decoded into several RISC86 operations.

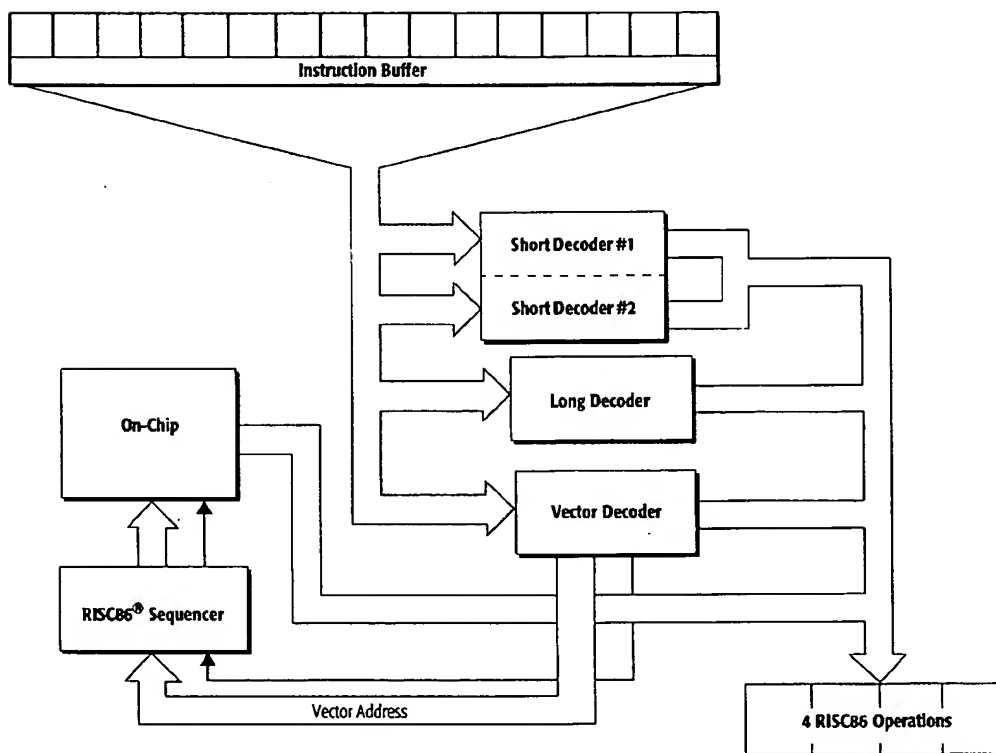


Figure 4. Processor Decode Logic

## 2 Internal Architecture

---

The AMD-K6 3D processor uses a combination of decoders to convert x86 instructions into RISC86 operations. The hardware consists of three sets of decoders—two parallel short decoders, one long decoder, and one vector decoder. The two parallel short decoders translate the most commonly-used x86 instructions (moves, shifts, branches, ALU, MMX, 3D, FPU) into zero, one, or two RISC86 operations each. The short decoders only operate on x86 instructions that are up to seven bytes long. In addition, they are designed to decode up to two x86 instructions per clock. The commonly-used x86 instructions that are greater than seven bytes but not more than 11 bytes long, and semi-commonly-used x86 instructions that are up to seven bytes long are handled by the long decoder.

The long decoder only performs one decode per clock and generates up to four RISC86 operations. All other translations (complex instructions, serializing conditions, interrupts and exceptions, etc.) are handled by a combination of the vector decoder and RISC86 operation sequences fetched from an on-chip ROM. For complex operations, the vector decoder logic provides the first set of RISC86 operations and a vector (initial ROM address) to a sequence of further RISC86 operations. The same types of RISC86 operations are fetched from the ROM as those that are generated by the hardware decoders.

*Note: Although all three sets of decoders are simultaneously fed a copy of the instruction buffer contents, only one of the three types of decoders is used during any one decode clock.*

The decoders or the on-chip RISC86 ROM always generate a group of four RISC86 operations. For decodes that cannot fill the entire group with four RISC86 operations, RISC86 NOP operations are placed in the empty locations of the grouping. For example, a long-decoded x86 instruction that converts to only three RISC86 operations is padded with a single RISC86 NOP operation and then passed to the scheduler. Up to six groups or 24 RISC86 operations can be placed in the scheduler at a time.

All of the common, and a few of the uncommon, floating-point instructions (also known as ESC instructions) are hardware decoded as short decodes. This decode generates a RISC86 floating-point operation and, optionally, an associated floating-point load or store operation. Floating-point or ESC instruction decode is only allowed in the first short decoder, but non-ESC instructions can be decoded simultaneously by the second short decoder along with an ESC instruction decode in the first short decoder.

All of the MMX and 3D instructions, with the exception of the EMMS, FEMMS, and PREFETCH instructions, are hardware decoded as short decodes. The MMX instruction decode generates a RISC86 MMX operation and, optionally, an associated MMX load or store operation. An 3D instruction decode generates a RISC86 3D operation and, optionally, an associated load or store operation. MMX and 3D instructions can be decoded in either or both of the short decoders.

## **2** *Internal Architecture*

### **Centralized Scheduler**

The scheduler is the heart of the AMD-K6 3D processor (see Figure 5 on page 17). It contains the logic necessary to manage out-of-order execution, data forwarding, register renaming, simultaneous issue and retirement of multiple RISC86 operations, and speculative execution. The scheduler's buffer can hold up to 24 RISC86 operations. This equates to a maximum of 12 x86 instructions. The scheduler can issue RISC86 operations from any of the 24 locations in the buffer. When possible, the scheduler can simultaneously issue a RISC86 operation to any available execution unit (store, load, branch, register X integer/MMX, register y integer/MMX, or floating-point). In total, the scheduler can issue up to six and retire up to four RISC86 operations per clock.

The main advantage of the scheduler and its operation buffer is the ability to examine an x86 instruction window equal to 12 x86 instructions at one time. This advantage is due to the fact that the scheduler operates on the RISC86 operations in parallel and allows the processor to perform dynamic on-the-fly instruction code scheduling for optimized execution. Although the scheduler can issue RISC86 operations for out-of-order execution, it always retires x86 instructions in order.

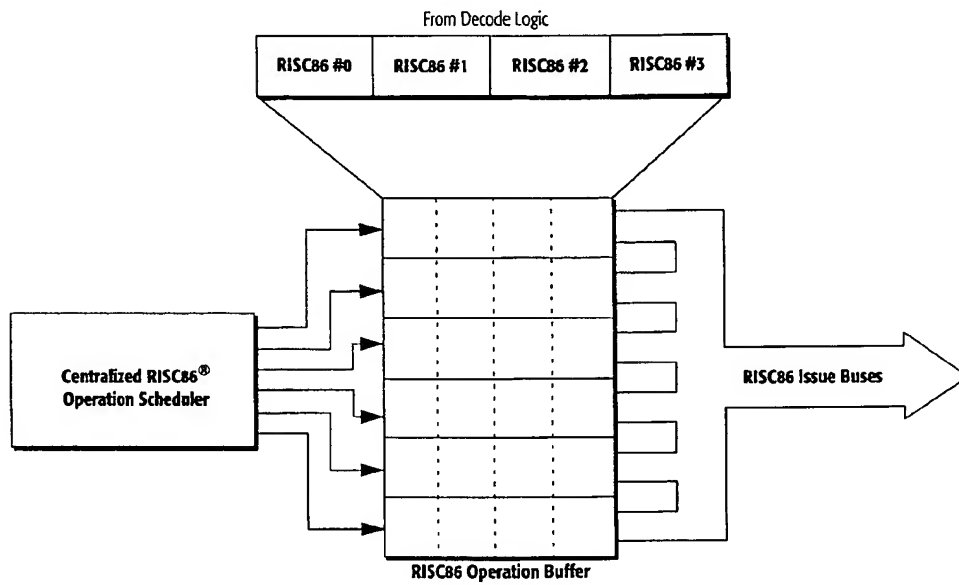


Figure 5. Processor Scheduler

## **2** *Internal Architecture*

### **Execution Units**

The AMD-K6 3D processor contains ten parallel execution units—store, load, integer X ALU, integer Y ALU, MMX ALU (X), MMX ALU (Y), MMX/3D multiplier, 3D ALU, floating-point, and branch condition. Each unit is independent and capable of handling the RISC86 operations. Table 1 on page 19 details the execution units, functions performed within these units, operation latency, and operation throughput. For more information, see “Execution Units and Dependency Latencies” on page 458 in Appendix B.

The store and load execution units are two-stage pipelined designs. The store unit performs data writes and register calculation for LEA/PUSH. Data memory and register writes from stores are available after one clock. Store operations are held in a store queue prior to execution. From there, they execute in order. The load unit performs data memory reads. Data is available from the load unit after two clocks. For more information, see “Load Unit” on page 462 and “Store Unit” on page 463, both in Appendix B.

The Integer X execution unit can operate on all ALU operations, multiplies, divides (signed and unsigned), shifts, and rotates.

The Integer Y execution unit can operate on the basic word and doubleword ALU operations—ADD, AND, CMP, OR, SUB, XOR, zero-extend and sign-extend operands.

**Table 1. Execution Latency and Throughput of Execution Units**

Functional Unit	Function	Latency	Throughput
Store	LEA/PUSH, Address (Pipelined)	1	1
	Memory Store (Pipelined)	1	1
Load	Memory Loads (Pipelined)	2	1
Integer X	Integer ALU	1	1
	Integer Multiply	2-3	2-3
	Integer Shift	1	1
Integer MMX	MMX ALU	1	1
	MMX Shifts, Packs, Unpack	1	1
	MMX Multiply	2	1
Integer Y	Basic ALU (16-bit and 32-bit operands)	1	1
Branch	Resolves Branch Conditions	1	1
FPU	FADD, FSUB, FMUL	2	2
3D	3D ALU	2	1
	3D Multiply	2	1
	3D Convert	2	1



## 2 Internal Architecture

### Register X and Y Pipelines

The MMX units and the 3D units share pipeline control with the Integer X and Integer Y units.

The register X and Y functional units are attached to the issue bus for the register X execution pipeline or the issue bus for the register Y execution pipeline or both. Figure 6 shows the details of the X and Y register pipelines. Each register pipeline has dedicated resources that consist of an integer execution unit and an MMX ALU execution unit, therefore allowing superscalar operation on integer and MMX instructions. In addition, both the X and Y issue buses are connected to the 3D ALU, the 3D/MMX multiplier and MMX shifter, which allows the appropriate RISC86 operation to be issued through either bus. For more information, see Figure 50 on page 91 in Chapter 4 and "Register Execution Units" on page 460 in Appendix B.

The branch condition unit is separate from the branch prediction logic in that it resolves conditional branches such as JCC and LOOP after the branch condition has been evaluated. For more information, see "Branch Condition Unit" on page 464 in Appendix B.

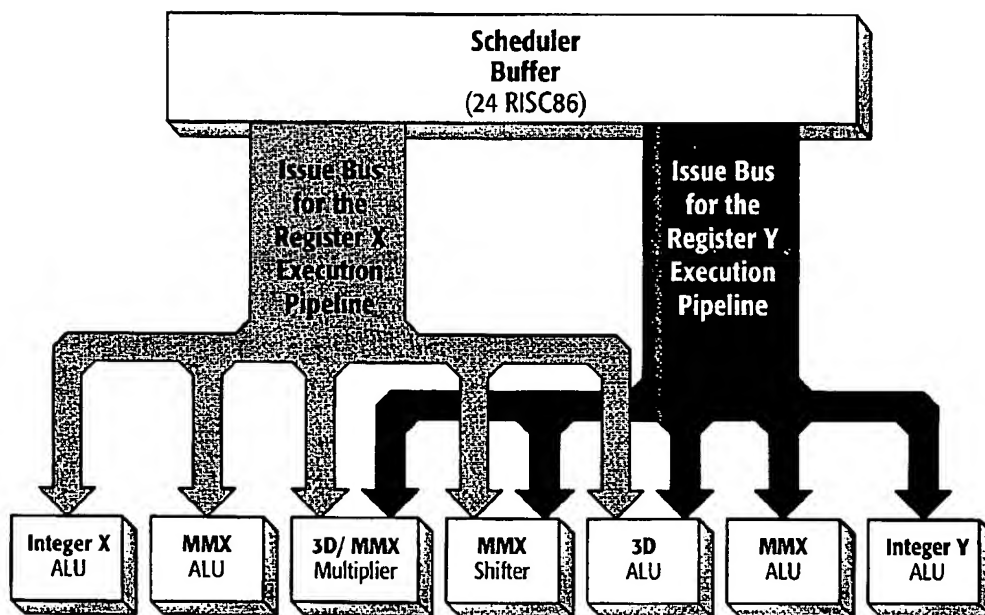


Figure 6. Register X and Y Functional Units

## **Branch-Prediction Logic**

Sophisticated branch logic that can minimize or hide the impact of changes in program flow is designed into the AMD-K6 3D processor. Branches in x86 code fit into two categories—unconditional branches, which always change program flow (that is, the branches are always taken) and conditional branches, which may or may not divert program flow (that is, the branches are taken or not-taken). When a conditional branch is not taken, the processor simply continues decoding and executing the next instructions in memory.

Typical applications have up to 10% of unconditional branches and another 10% to 20% conditional branches. The processor branch logic has been designed to handle this type of program behavior and its negative effects on instruction execution, such as stalls due to delayed instruction fetching and the draining of the processor pipeline. The branch logic contains an 8192-entry branch history table, a 16-entry by 16-byte branch target cache, a 16-entry return address stack, and a branch execution unit.

### **Branch History Table**

The AMD-K6 3D processor handles unconditional branches without any penalty by redirecting instruction fetching to the target address of the unconditional branch. However, conditional branches require the use of the dynamic branch-prediction mechanism built into the processor. A two-level adaptive history algorithm is implemented in an 8192-entry branch history table. This table stores executed branch information, predicts individual branches, and predicts the behavior of groups of branches. To accommodate the large branch history table, the processor does not store predicted target addresses. Instead, the branch target addresses are calculated on-the-fly using ALUs during the decode stage. The adders calculate all possible target addresses before the instructions are fully decoded and the processor chooses which addresses are valid.

### **Branch Target Cache**

To avoid a one clock cache-fetch penalty when a branch is predicted taken, a built-in branch target cache supplies the first 16 bytes of instructions directly to the instruction buffer (assuming the target address hits this cache). (See Figure 3 on page 12.) The branch target cache is organized as 16 entries of 16 bytes. In total, the branch prediction logic achieves branch prediction rates greater than 95%.

## **2** *Internal Architecture*

### **Return Address Stack**

The return address stack is a special device designed to optimize CALL and RET pairs. Software is typically compiled with subroutines that are frequently called from various places in a program. This is usually done to save space. Entry into the subroutine occurs with the execution of a CALL instruction. At that time, the processor pushes the address of the next instruction in memory following the CALL instruction onto the stack (allocated space in memory). When the processor encounters a RET instruction (within or at the end of the subroutine), the branch logic pops the address from the stack and begins fetching from that location. To avoid the latency of main memory accesses during CALL and RET operations, the return address stack caches the pushed addresses.

### **Branch Execution Unit**

The branch execution unit enables efficient speculative execution. This unit gives the processor the ability to execute instructions beyond conditional branches before knowing whether the branch prediction was correct. The processor does not permanently update the x86 registers or memory locations until all speculatively executed conditional branch instructions are resolved. When a prediction is incorrect, the processor backs out to the point of the mispredicted branch instruction and restores all registers. The AMD-K6 3D processor can support up to seven outstanding branches.

# 3

## Software Environment

This chapter provides a general overview of the AMD-K6 3D processor's x86 software environment and briefly describes the data types, registers, operating modes, interrupts, and instructions supported by the AMD-K6 3D architecture and design implementation.

### Registers

---

The AMD-K6 3D processor contains all the registers defined by the x86 architecture, including general-purpose, segment, floating-point, MMX (3D), EFLAGS, control, task, debug, test, and descriptor/memory-management registers. In addition, this chapter provides information on the processor model-specific registers (MSRs).

*Note: Areas of the register designated as Reserved should not be modified by software.*

### General-Purpose Registers

The eight 32-bit x86 general-purpose registers are used to hold integer data or memory pointers used by instructions. Table 2 on page 24 contains a list of the general-purpose registers and the functions for which they are used.

# 3 Software Environment

Table 2. General-Purpose Registers

Register	Function
EAX	Commonly used as an accumulator
EBX	Commonly used as a pointer
ECX	Commonly used for counting in loop operations
EDX	Commonly used to hold I/O information and to pass parameters
EDI	Commonly used as a destination pointer by the ES segment
ESI	Commonly used as a source pointer by the DS segment
ESP	Used to point to the stack segment
EBP	Used to point to data within the stack segment

In order to support byte and word operations, EAX, EBX, ECX, and EDX can also be used as 8-bit and 16-bit registers. The shorter registers are overlaid on the longer ones. For example, the name of the 16-bit version of EAX is AX (low 16 bits of EAX) and the 8-bit names for AX are AH (high order bits) and AL (low order bits). The same naming convention applies to EBX, ECX, and EDX. EDI, ESI, ESP, and EBP can be used as smaller 16-bit registers called DI, SI, SP, and BP respectively, but these registers do not have 8-bit versions. Figure 7 shows the EAX register with its name components, and Table 3 on page 25 lists the doubleword (32-bit) general-purpose registers and their corresponding word (16-bit) and byte (8-bit) versions.

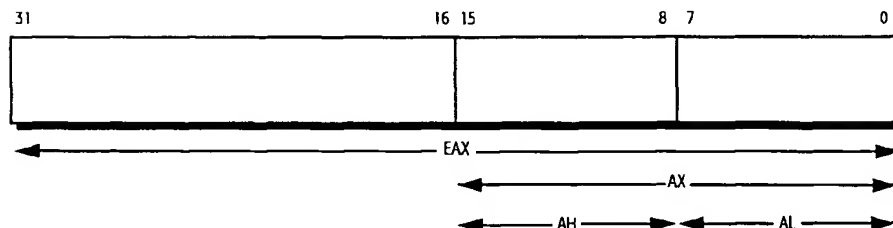


Figure 7. EAX Register with 16-Bit and 8-Bit Name Components

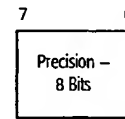
Table 3. General-Purpose Register Dword, Word, and Byte Names

32-Bit Name (Dword)	16-Bit Name (Word)	8-Bit Name (High-order Bits)	8-Bit Name (Low-order Bits)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
EDI	DI	—	—
ESI	SI	—	—
ESP	SP	—	—
EBP	BP	—	—

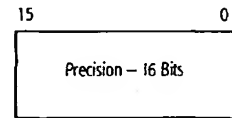
## Integer Data Types

Four types of data are used in general-purpose registers—byte, word, doubleword, and quadword integers. Figure 8 shows the format of the integer data registers.

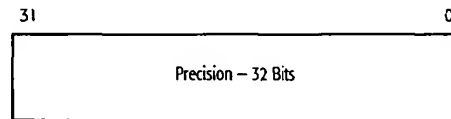
Byte Integer



Word Integer



Doubleword Integer



Quadword Integer

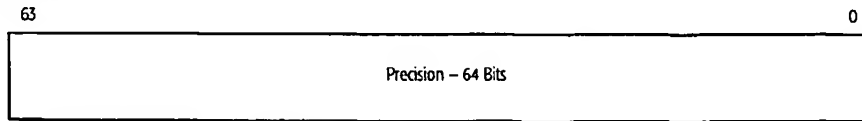


Figure 8. Integer Data Registers

## 3 Software Environment

### Segment Registers

The six 16-bit segment registers are used as pointers to areas (segments) of memory. Table 4 lists the segment registers and their functions. Figure 9 shows the format for all six segment registers.

Table 4. Segment Registers

Segment Register	Segment Register Function
CS	Code segment, where instructions are located
DS	Data segment, where data is located
ES	Data segment, where data is located
FS	Data segment, where data is located
GS	Data segment, where data is located
SS	Stack segment

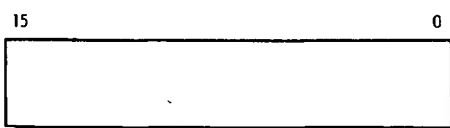


Figure 9. Segment Register

### Segment Usage

The operating system determines the type of memory model that is implemented. The segment register usage is determined by the operating system's memory model. In a Real mode memory model the segment register points to the base address in memory. In a Protected mode memory model the segment register is called a selector and it selects a segment descriptor in a descriptor table. This descriptor contains a pointer to the base of the segment, the limit of the segment, and various protection attributes. For more information on descriptor formats, see "Descriptors and Gates" on page 48. Figure 10 on page 27 shows segment usage for Real mode and Protected mode memory models.

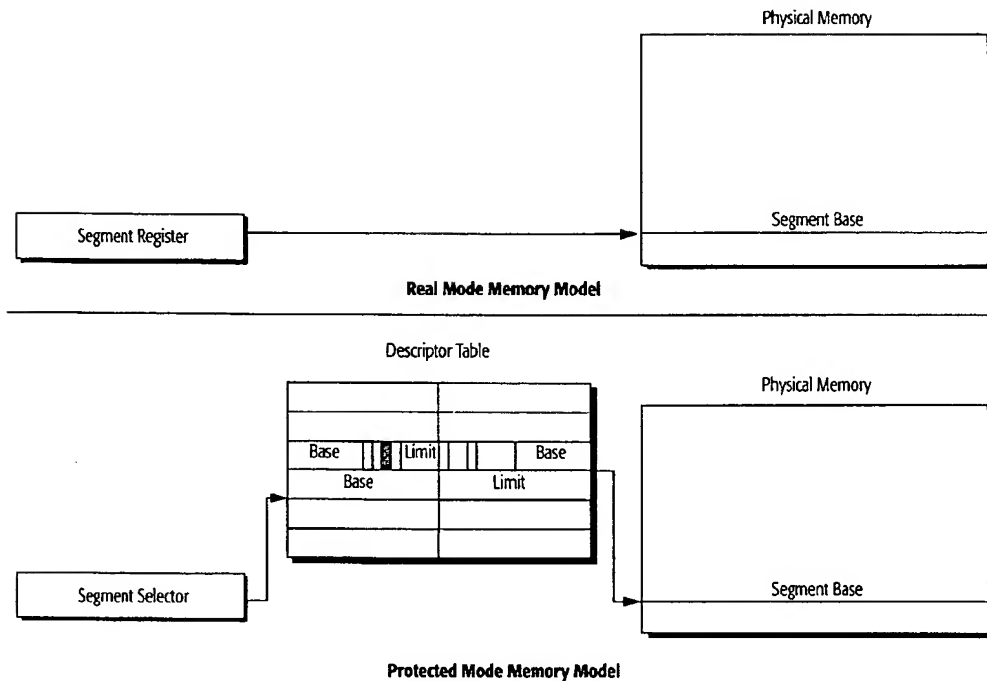


Figure 10. Segment Usage

## Instruction Pointer

The instruction pointer (EIP or IP) is used in conjunction with the code segment register (CS). The instruction pointer is either a 32-bit register (EIP) or a 16-bit register (IP) that keeps track of where the next instruction resides within memory. This register cannot be directly manipulated, but can be altered by modifying return pointers when a JMP or CALL instruction is used.

## Floating-Point Registers

The floating-point execution unit in the AMD-K6 3D processor is designed to perform mathematical operations on non-integer numbers. This floating-point unit conforms to the IEEE 754 and 854 standards and uses several registers to meet these standards—eight numeric floating-point registers, a status word register, a control word register, and a tag word register.



# 3 Software Environment

The eight floating-point registers are physically 80 bits wide and labeled FPR0–FPR7. Figure 11 shows the format of these floating-point registers. For more information, see Chapter 9, “Floating-Point and Multimedia Execution Units” on page 253. See “Floating-Point Register Data Types” on page 30 for information on allowable floating-point data types.

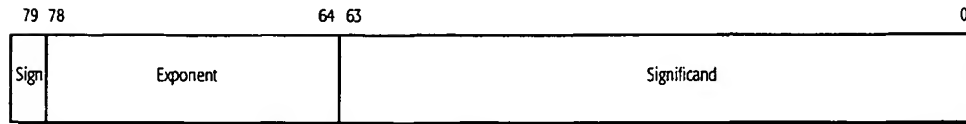


Figure 11. Floating-Point Register

The 16-bit FPU status word register contains information about the state of the floating-point unit. Figure 12 shows the format of this register.

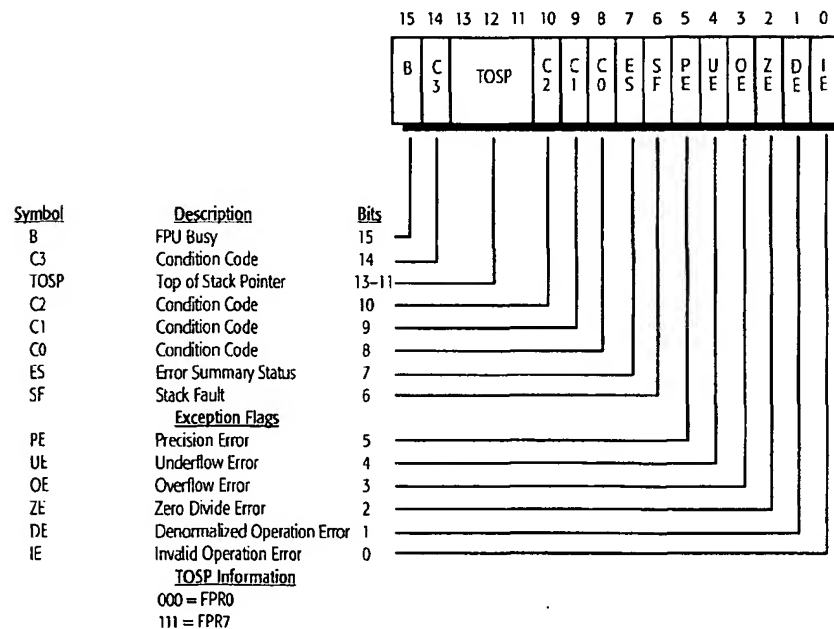


Figure 12. FPU Status Word Register

The FPU control word register allows a programmer to manage the FPU processing options. Figure 13 shows the format of this register.

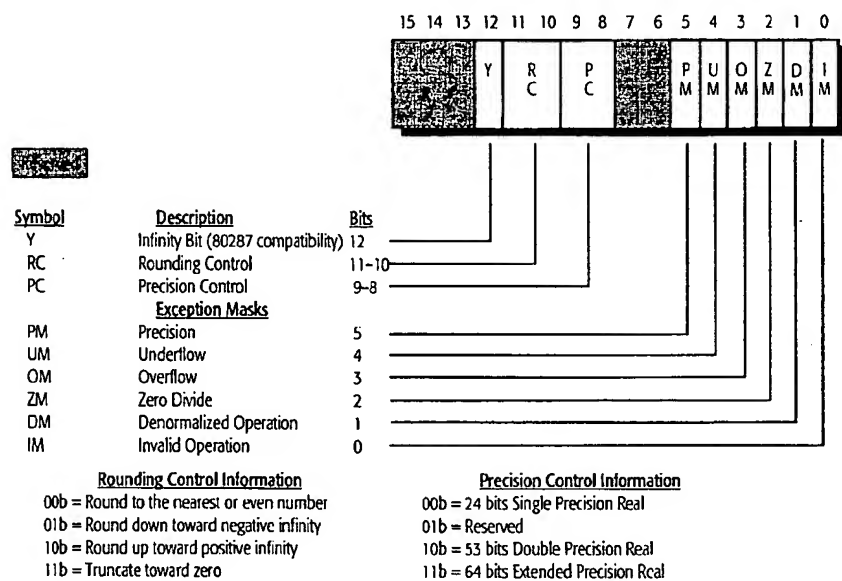


Figure 13. FPU Control Word Register

The FPU tag word register contains information about the registers in the register stack. Figure 14 shows the format of this register.

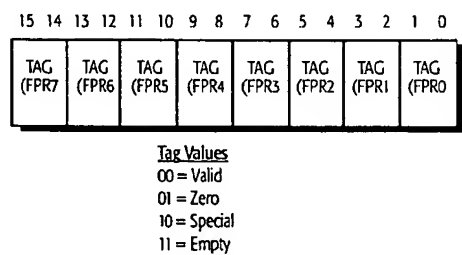


Figure 14. FPU Tag Word Register

# 3 Software Environment

## Floating-Point Register Data Types

Floating-point registers use four different types of data—packed decimal, single-precision real, double-precision real, and extended-precision real. Figures 15 and 16 show the formats for these registers. For more information, see Chapter 9, “Floating-Point and Multimedia Execution Units” on page 253

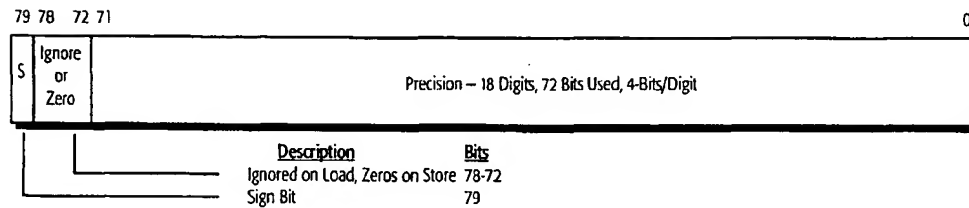
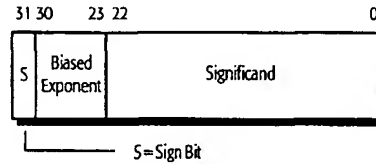
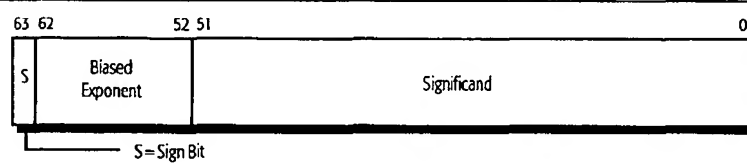


Figure 15. Packed Decimal Data Register

### Single-Precision Real



### Double-Precision Real



### Extended-Precision Real

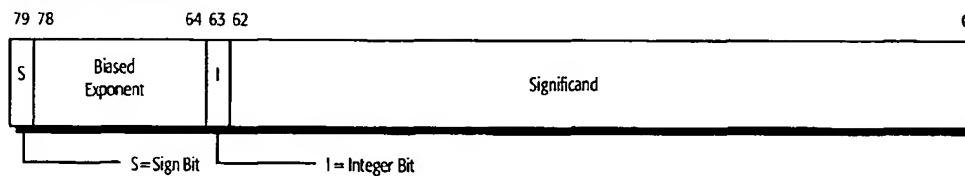


Figure 16. Precision Real Data Registers

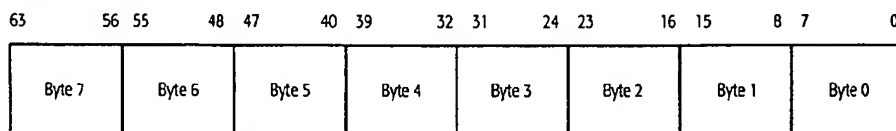
## MMX/3D Registers

The AMD-K6 3D processor implements eight 64-bit MMX/3D registers for use by multimedia software. These registers are mapped on the floating-point register stack. The 3D and MMX instructions refer to these registers as mmreg0 to mmreg7. Figure 46 on page 84 shows the format of these registers. For more information, see “3D Register Set” on page 84 and “MMX Register Set” on page 350 in Appendix A.

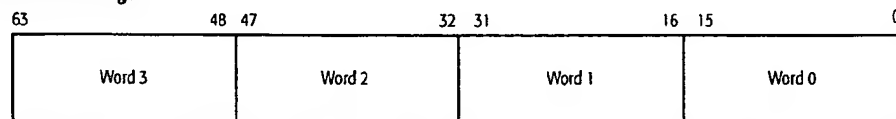
## MMX Data Types

For the MMX instructions, the MMX registers use three types of data—packed eight-byte integer, packed quadword integer, and packed dual doubleword integer. Figure 17 shows the format of these data types. For more information, see “MMX Data Type Details” on page 352 in Appendix A.

### Packed Bytes Integer



### Packed Words Integer



### Packed Doubleword Integer

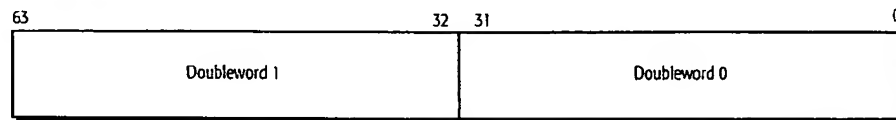


Figure 17. MMX Data Types

## 3 Software Environment

### 3D Data Types

For 3D instructions, the MMX registers use packed single-precision real data. Figure 18 shows the format of the 3D data type. For more information, see “3D Data Type Details” on page 85.

#### Packed Single Precision Floating Point

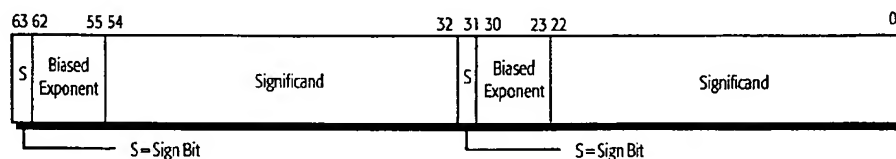


Figure 18. 3D Data Types

EFLAGS Register

The EFLAGS register provides for three different types of flags—system, control, and status. The system flags provide operating system controls, the control flag provides directional information for string operations, and the status flags provide information resulting from logical and arithmetic operations. Figure 19 shows the format of this register.

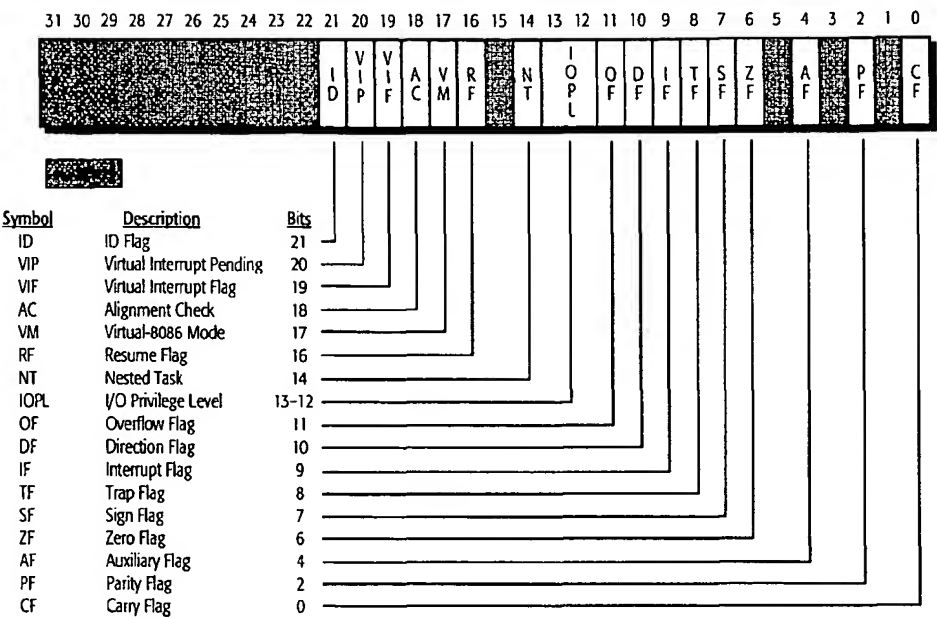


Figure 19. EFLAGS Registers

# 3 Software Environment

## Control Registers

The five control registers contain system control bits and pointers. Figures 20 through 24 show the formats of these registers.

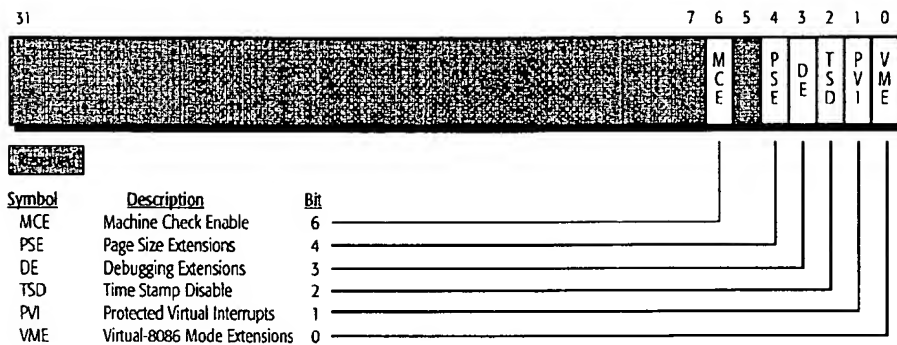


Figure 20. Control Register 4 (CR4)

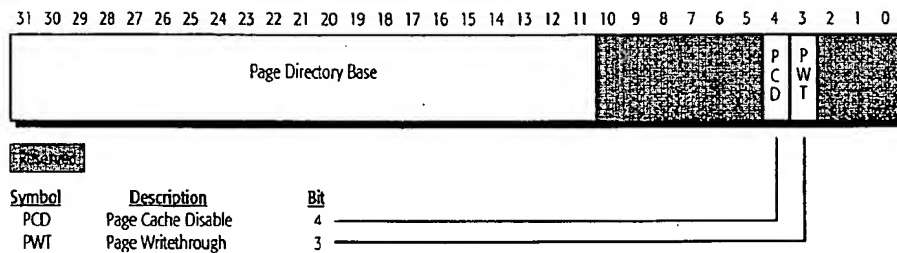


Figure 21. Control Register 3 (CR3)

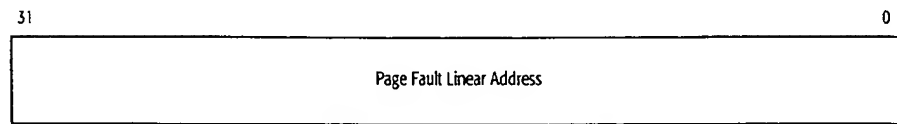
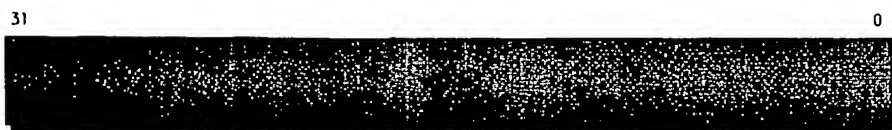
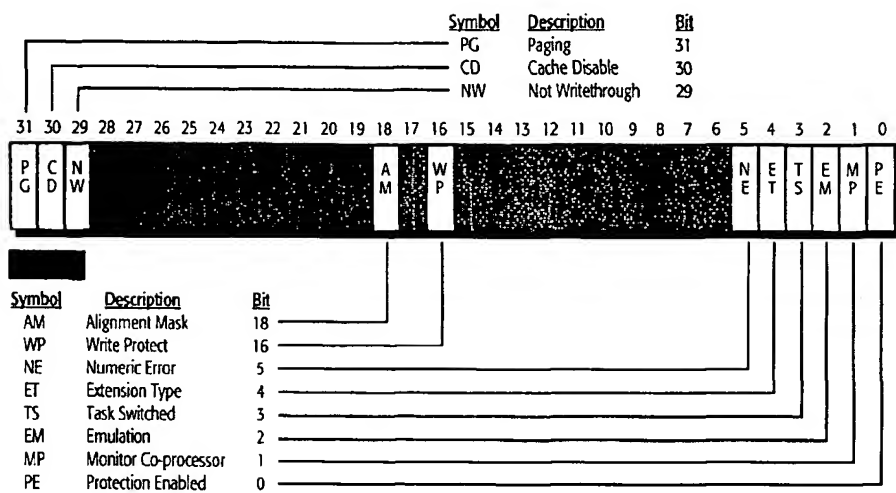


Figure 22. Control Register 2 (CR2)



**Figure 23. Control Register 1 (CR1)**



**Figure 24. Control Register 0 (CR0)**



# 3 Software Environment

## Debug Registers

Figures 25 through 28 show the 32-bit debug registers supported by the processor.

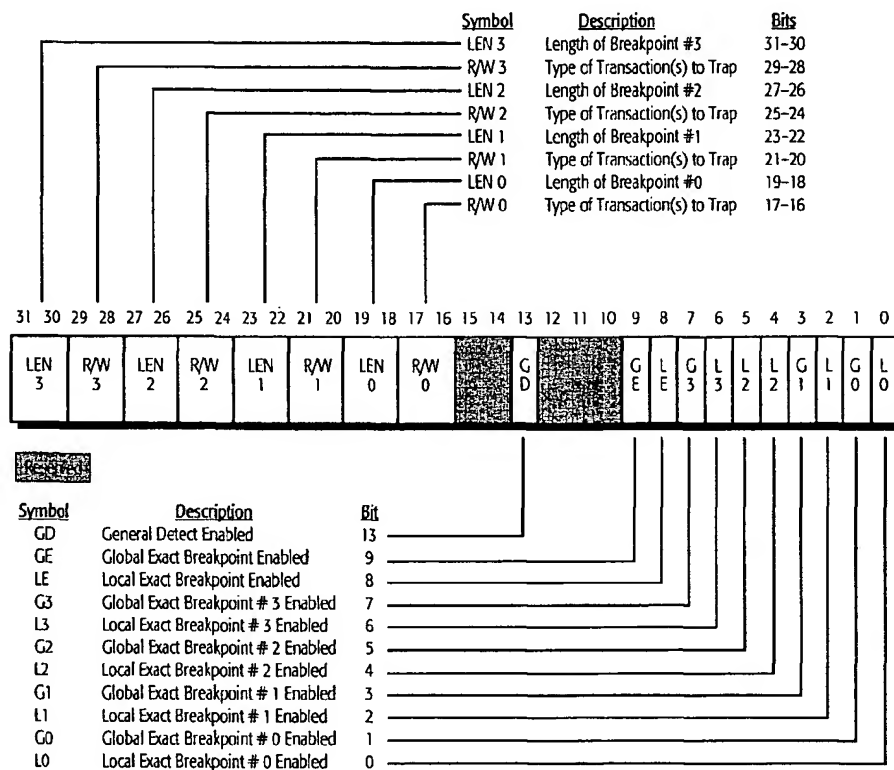
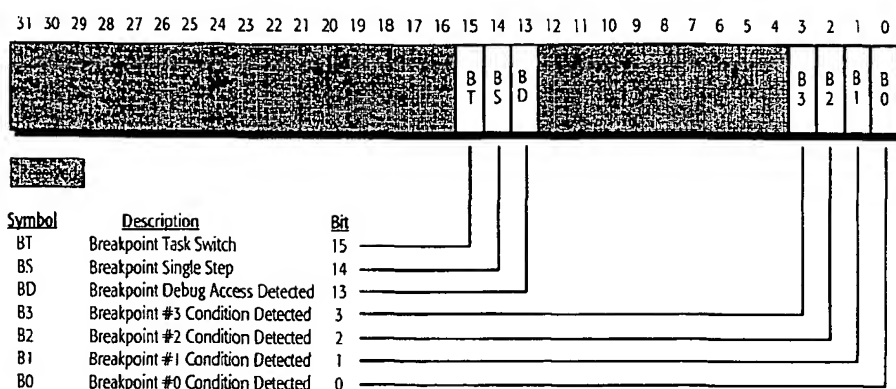
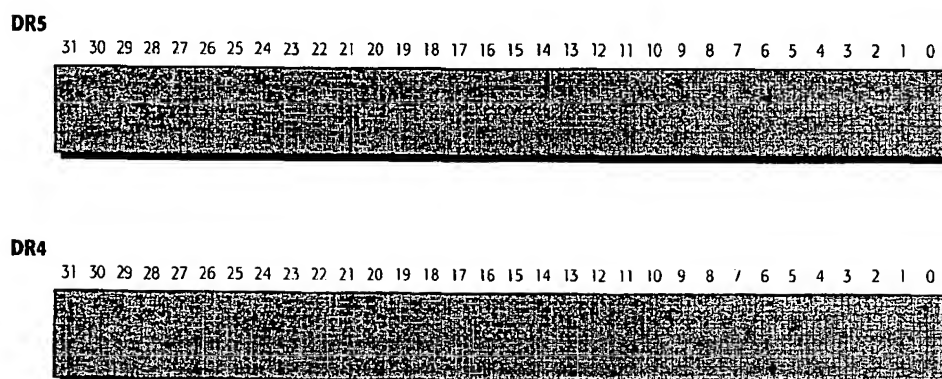


Figure 25. Debug Register DR7



**Figure 26. Debug Register DR6**



**Figure 27. Debug Registers DR5 and DR4**

# 3 Software Environment

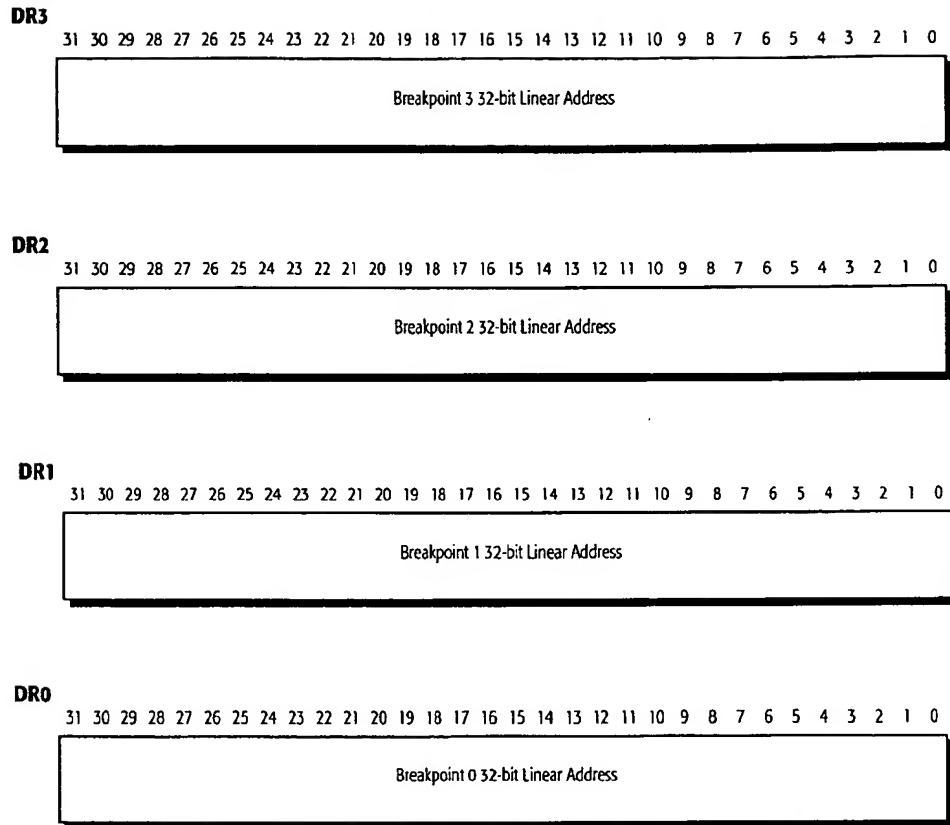


Figure 28. Debug Registers DR3, DR2, DR1, and DR0

## Model-Specific Registers (MSR)

The AMD-K6 3D processor provides seven MSRs. The value in the ECX register selects the MSR to be addressed by the RDMSR and WRMSR instructions. The values in EAX and EDX are used as inputs and outputs by the RDMSR and WRMSR instructions. Table 5 lists the MSRs and the corresponding value of the ECX register. Figures 29 through 35 show the MSR formats.

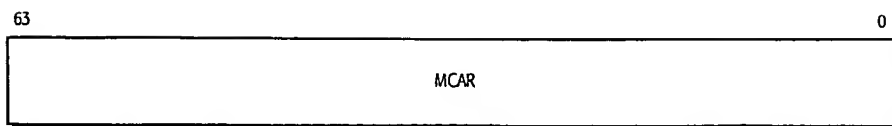
**Table 5. Model-Specific Registers (MSRs)**

Model-Specific Register	Value of ECX
Machine Check Address Register (MCAR)	00h
Machine Check Type Register (MCTR)	01h
Test Register 12 (TR12)	0Eh
Time Stamp Counter (TSC)	10h
Extended Feature Enable Register (EFER)	C000_0080h
SYSCALL/SYSRET Target Address Register (STAR)	C000_0081h
Write Handling Control Register (WHCR)	C000_0082h

For more information about the RDMSR and WRMSR instructions, see the *AMD K86™ Family BIOS and Software Tools Development Guide*, document number 21062.

### MCAR and MCTR

The processor does not support the generation of a machine check exception. However, the processor does provide a 64-bit machine check address register (MCAR), a 64-bit machine check type register (MCTR), and a machine check enable (MCE) bit in CR4. Because the processor does not support machine check exceptions, the contents of the MCAR and MCTR are only affected by the WRMSR instruction and by RESET being sampled asserted (where all bits in each register are reset to 0).



**Figure 29. Machine-Check Address Register (MCAR)**

# 3 Software Environment



Figure 30. Machine-Check Type Register (MCTR)

**Test Register 12 (TR12)** Test register 12 provides a method for disabling the L1 caches. Figure 31 shows the format of TR12.

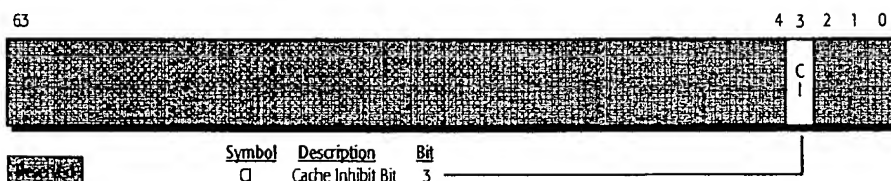


Figure 31. Test Register 12 (TR12)

**Time Stamp Counter** With each processor clock cycle, the processor increments the 64-bit time stamp counter (TSC) MSR. Figure 32 shows the format of the TSC.

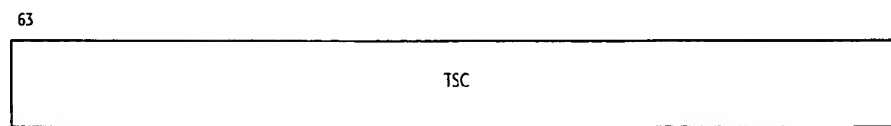


Figure 32. Time Stamp Counter (TSC)

## Extended Feature Enable Register (EFER)

The extended feature enable register (EFER) contains the control bits that enable the extended features of the AMD-K6 3D processor. Figure 33 shows the format of the EFER register, and Table 6 defines the function of each bit in the EFER register.

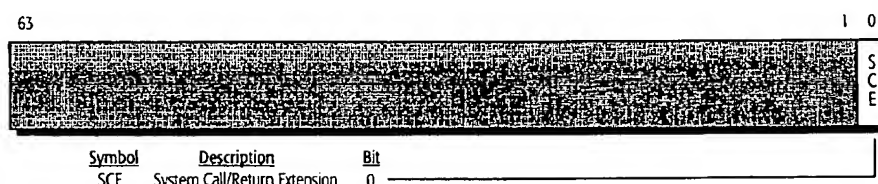


Figure 33. Extended Feature Enable Register (EFER)

Table 6. Extended Feature Enable Register (EFER) Definition

Bit	Description	R/W
63-1	Reserved	R
0	System Call Extension (SCE)	R/W

## SYSCALL/SYSRET Target Address Register (STAR)

The SYSCALL/SYSRET target address register (STAR) contains the target EIP address used by the SYSCALL instruction and the 16-bit code and stack segment selector bases used by the SYSCALL and SYSRET instructions. Figure 34 shows the format of the STAR register, and Table 7 on page 42 defines the function of each bit of the STAR register. For more information, see the *SYSCALL and SYSRET Instruction Specification Application Note*, document number 21086.

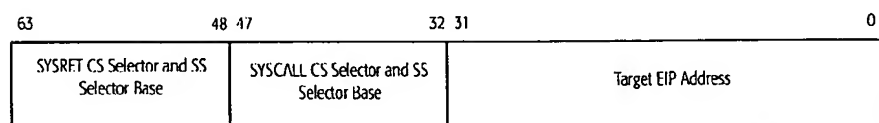


Figure 34. SYSCALL/SYSRET Target Address Register (STAR)

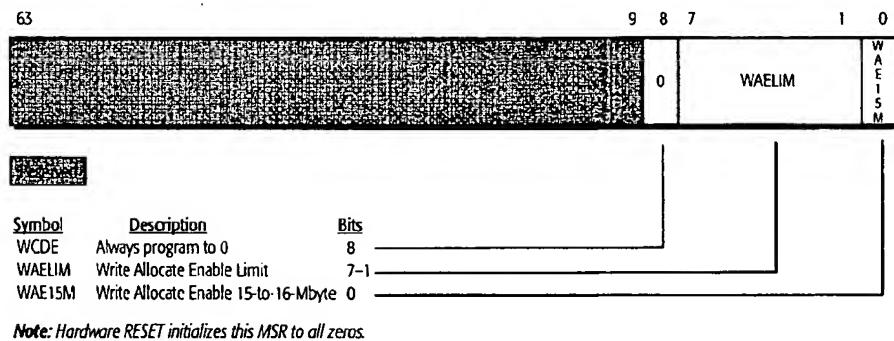
# 3 Software Environment

**Table 7. SYSCALL/SYSRET Target Address Register (STAR) Definition**

Bit	Description	R/W
63–48	SYSRET CS and SS Selector Base	R/W
47–32	SYSCALL CS and SS Selector Base	R/W
31–0	Target EIP Address	R/W

## Write Handling Control Register (WHCR)

The write handling control register (WHCR) is a MSR that contains three fields—the WCDE bit, write allocate enable limit (WAE LIM) field, and the write allocate enable 15-to-16-Mbyte (WAE15M) bit. Figure 35 shows the format of WHCR. See “Write Allocate” on page 240 for more information.



**Figure 35. Write Handling Control Register (WHCR)**

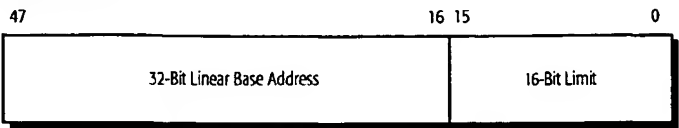
## Memory Management Registers

The AMD-K6 3D processor controls segmented memory management with the registers listed in Table 8. Figure 36 on page 43 shows the formats of these registers.

**Table 8. Memory Management Registers**

Register Name	Function
Global Descriptor Table Register	Contains a pointer to the base of the global descriptor table
Interrupt Descriptor Table Register	Contains a pointer to the base of the interrupt descriptor table
Local Descriptor Table Register	Contains a pointer to the local descriptor table of the current task
Task Register	Contains a pointer to the task state segment of the current task

Global and Interrupt Descriptor Table Registers



Local Descriptor Table Register and Task Register

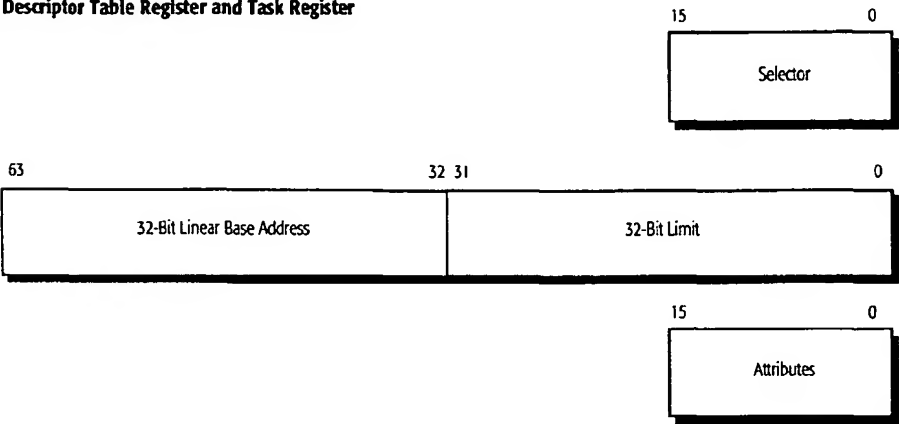


Figure 36. Memory Management Registers



# 3 Software Environment

## Task State Segment

Figure 37 shows the format of the task state segment (TSS).

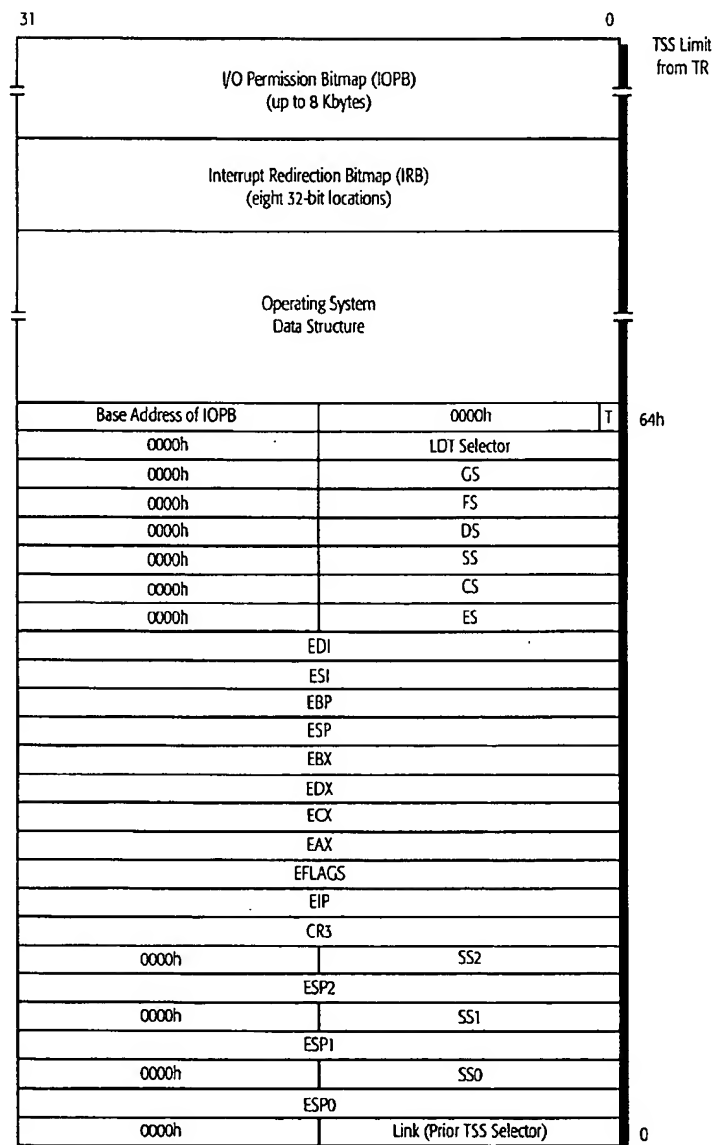


Figure 37. Task State Segment (TSS)

## Paging

The AMD-K6 3D processor can physically address up to four Gbytes of memory. This memory can be segmented into pages. The size of these pages is determined by the operating system design and the values set up in the page directory entries (PDE) and page table entries (PTE). The processor can access both 4-Kbyte pages and 4-Mbyte pages, and the page sizes can be intermixed within a page directory. When the page size extension (PSE) bit in CR4 is set, the processor translates linear addresses using either the 4-Kbyte translation lookaside buffer (TLB) or the 4-Mbyte TLB, depending on the state of the page size (PS) bit in the page directory entry. Figures 38 and 39 show how 4-Kbyte and 4-Mbyte page translations work.

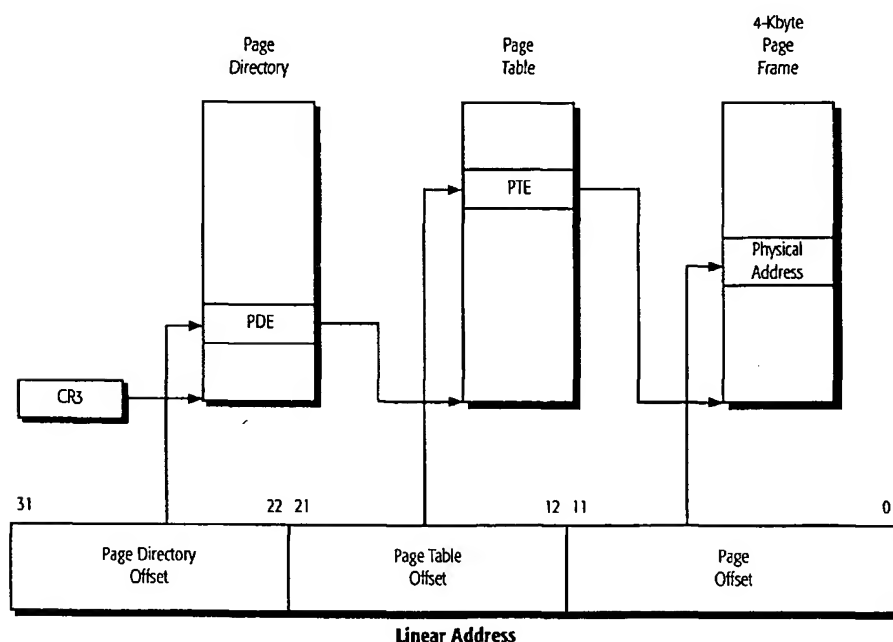


Figure 38. 4-Kbyte Paging Mechanism

### 3 Software Environment

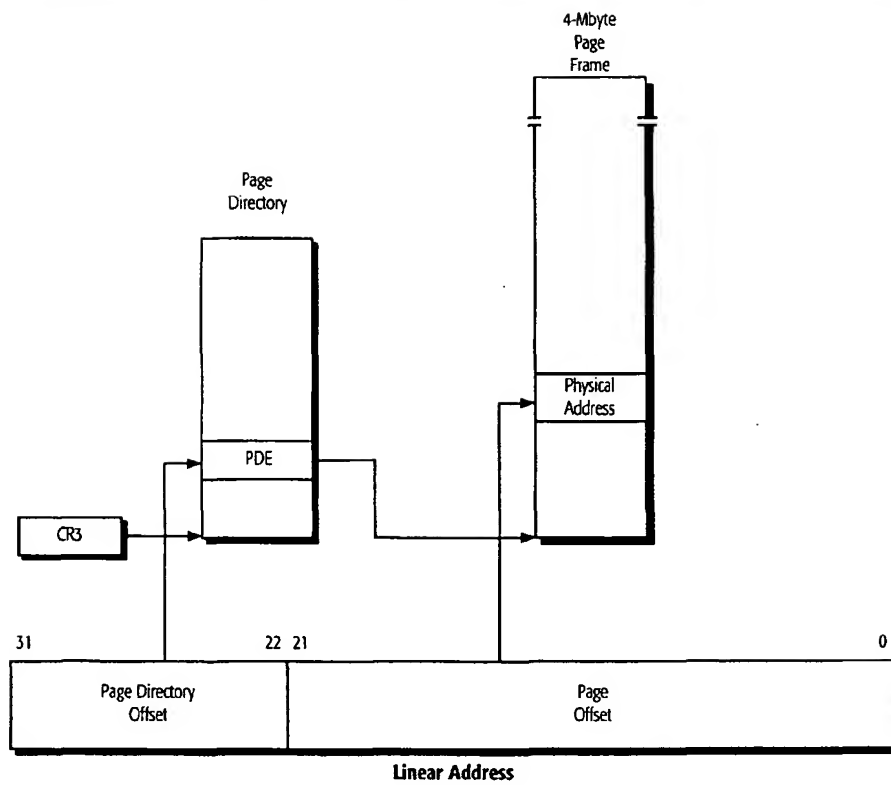


Figure 39. 4-Mbyte Paging Mechanism

Figures 40 through 42 show the formats of the PDE and PTE. These entries contain information regarding the location of pages and their status.

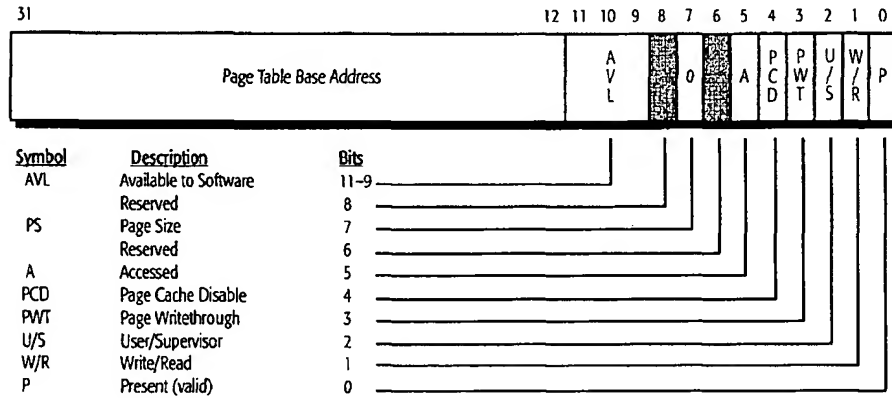


Figure 40. Page Directory Entry 4-Kbyte Page Table (PDE)

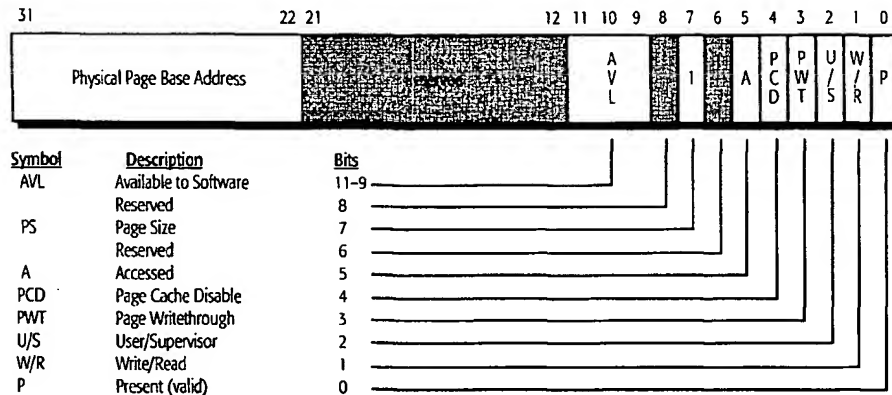


Figure 41. Page Directory Entry 4-Mbyte Page Table (PDE)

# 3 Software Environment

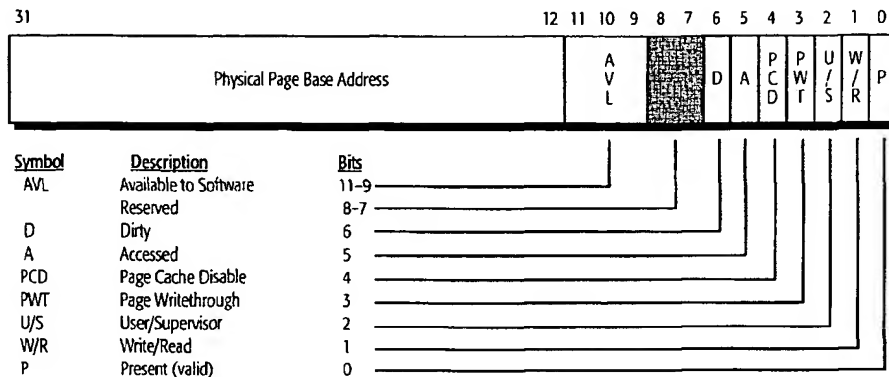


Figure 42. Page Table Entry (PTE)

## Descriptors and Gates

There are various types of structures and registers in the x86 architecture that define, protect, and isolate code segments, data segments, task state segments, and gates. These structures are called descriptors.

Figure 43 on page 49 shows the application segment descriptor format. Table 9 on page 49 contains information describing the memory segment type to which the descriptor points. The application segment descriptor is used to point to either a data or code segment.

Figure 44 on page 50 shows the system segment descriptor format. Table 10 on page 50 contains information describing the type of segment or gate to which the descriptor points. The system segment descriptor is used to point to a task state segment, a call gate, or a local descriptor table.

The AMD-K6 3D processor uses gates to transfer control between executable segments with different privilege levels. Figure 45 on page 51 shows the format of the gate descriptor types. Table 10 contains information describing the type of segment or gate to which the descriptor points.

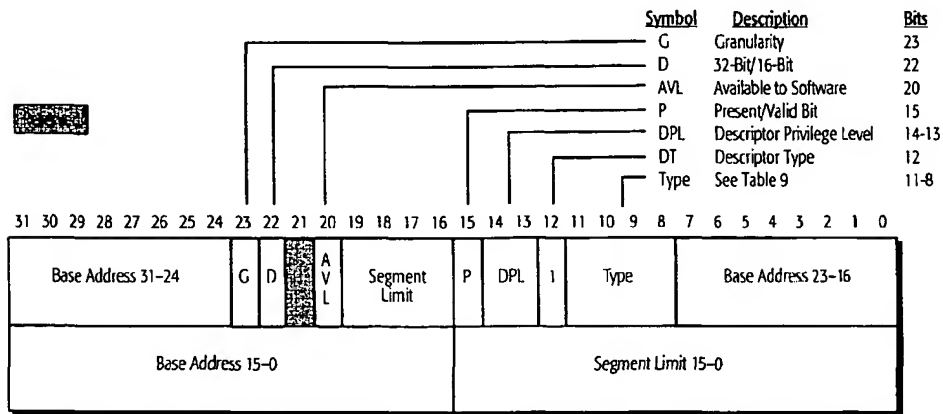


Figure 43. Application Segment Descriptor

Table 9. Application Segment Types

Type	Data/Code	Description
0	Data	Read-Only
1		Read-Only—Accessed
2		Read/Write
3		Read/Write—Accessed
4		Read-Only—Expand-down
5		Read-Only—Expand-down, Accessed
6		Read/Write—Expand-down
7		Read/Write—Expand-down, Accessed
8	Code	Execute-Only
9		Execute-Only—Accessed
A		Execute/Read
B		Execute/Read—Accessed
C		Execute-Only—Conforming
D		Execute-Only—Conforming, Accessed
E		Execute/Read-Only—Conforming
F		Execute/Read-Only—Conforming, Accessed

# 3 Software Environment

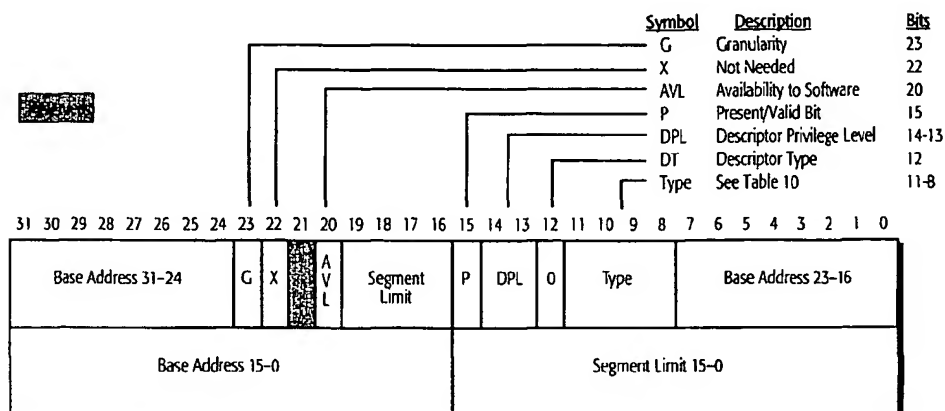


Figure 44. System Segment Descriptor

Table 10. System Segment and Gate Types

Type	Description
0	Reserved
1	Available 16-bit TSS
2	LDT
3	Busy 16-bit TSS
4	16-bit Call Gate
5	Task Gate
6	16-bit Interrupt Gate
7	16-bit Trap Gate
8	Reserved
9	Available 32-bit TSS
A	Reserved
B	Busy 32-bit TSS
C	32-bit Call Gate
D	Reserved
E	32-bit Interrupt Gate
F	32-bit Trap Gate

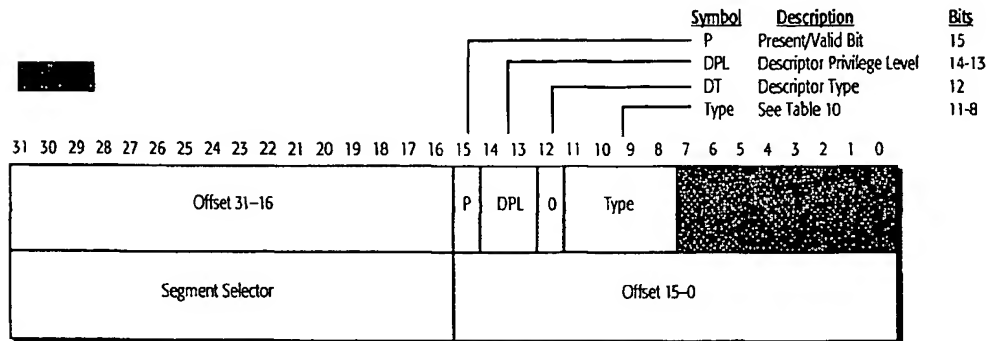


Figure 45. Gate Descriptor

## Exceptions and Interrupts

Table 11 summarizes the exceptions and interrupts.

Table 11. Summary of Exceptions and Interrupts

Interrupt Number	Interrupt Type	Cause
0	Divide by Zero Error	DIV, IDIV
1	Debug	Debug trap or fault
2	Non-Maskable Interrupt	NMI signal sampled asserted
3	Breakpoint	Int 3
4	Overflow	INTO
5	Bounds Check	BOUND
6	Invalid Opcode	Invalid instruction
7	Device Not Available	ESC and WAIT
8	Double Fault	Fault occurs while handling a fault
9	Reserved - Interrupt 13	—
10	Invalid TSS	Task switch to an invalid segment
11	Segment Not Present	Instruction loads a segment and present bit is 0 (invalid segment)
12	Stack Segment	Stack operation causes limit violation or present bit is 0
13	General Protection	Segment related or miscellaneous invalid actions
14	Page Fault	Page protection violation or a reference to missing page
16	Floating-Point Error	Arithmetic error generated by floating-point instruction
17	Alignment Check	Data reference to an unaligned operand. (The AC flag and the AM bit of CR0 are set to 1.)
0-255	Software Interrupt	INT n



# 3 Software Environment

## Instructions Supported by the Processor

This section documents all of the x86 instructions supported by the AMD-K6 3D processor. The following tables show the instruction mnemonic, opcode, modR/M byte, decode type, and RISC86 operation(s) for each instruction. Tables 12 through 15 define the integer, floating-point, MMX, 3D instructions, and new instructions for the processor, respectively.

The first column in these tables indicates the instruction mnemonic and operand types with the following notations:

- *reg8*—byte integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte
- *mreg8*—byte integer register or byte integer value in memory defined by the modR/M byte
- *reg16/32*—word or doubleword integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte
- *mreg16/32*—word or doubleword integer register, or word or doubleword integer value in memory defined by the modR/M byte
- *mem8*—byte integer value in memory
- *mem16/32*—word or doubleword integer value in memory
- *mem32/48*—doubleword or 48-bit integer value in memory
- *mem48*—48-bit integer value in memory
- *mem64*—64-bit value in memory
- *imm8*—8-bit immediate value
- *imm16/32*—16-bit or 32-bit immediate value
- *disp8*—8-bit displacement value
- *disp16/32*—16-bit or 32-bit displacement value
- *disp32/48*—doubleword or 48-bit displacement value
- *eXX*—register width depending on the operand size
- *mem32real*—32-bit floating-point value in memory
- *mem64real*—64-bit floating-point value in memory
- *mem80real*—80-bit floating-point value in memory
- *mmreg*—MMX/3D register
- *mmreg1*—MMX/3D register defined by bits 5, 4, and 3 of the modR/M byte
- *mmreg2*—MMX/3D register defined by bits 2, 1, and 0 of the modR/M byte

The second and third columns list all applicable opcode bytes.

The fourth column lists the modR/M byte when used by the instruction. The modR/M byte defines the instruction as a register or memory form. If modR/M bits 7 and 6 are documented as mm (memory form), mm can only be 10b, 01b or 00b.

The fifth column lists the type of instruction decode—short, long, and vector. The processor decode logic can process two short, one long, or one vector decode per clock.

The sixth column lists the type of RISC86 operation(s) required for the instruction. The operation types and corresponding execution units are as follows:

- *load, fload, mload*—load unit
- *store, fstore, mstore*—store unit
- *alu*—either of the integer execution units
- *alux*—integer X execution unit only
- *branch*—branch condition unit
- *float*—floating-point execution unit
- *mmx*—MMX execution unit for multimedia software
- *3D*—3D instructions execution unit
- *limm*—load immediate, instruction control unit

**Table 12. Integer Instructions**

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
AAA	37h			vector	
AAD	D5h	0Ah		vector	
AAM	D4h	0Ah		vector	
AAS	3Fh			vector	
ADC mreg8, reg8	10h		11-xxx-xxx	short	alux
ADC mem8, reg8	10h		mm-xxx-xxx	long	load, alux, store
ADC mreg16/32, reg16/32	11h		11-xxx-xxx	short	alu
ADC mem16/32, reg16/32	11h		mm-xxx-xxx	long	load, alu, store
ADC reg8, mreg8	12h		11-xxx-xxx	short	alux
ADC reg8, mem8	12h		mm-xxx-xxx	short	load, alux
ADC reg16/32, mreg16/32	13h		11-xxx-xxx	short	alu
ADC reg16/32, mem16/32	13h		mm-xxx-xxx	short	load, alu

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
ADC AL, imm8	14h		xx-xxx-xxx	short	alux
ADC EAX, imm16/32	15h		xx-xxx-xxx	short	alu
ADC mreg8, imm8	80h		11-010-xxx	short	alux
ADC mem8, imm8	80h		mm-010-xxx	long	load, alux, store
ADC mreg16/32, imm16/32	81h		11-010-xxx	short	alu
ADC mem16/32, imm16/32	81h		mm-010-xxx	long	load, alu, store
ADC mreg16/32, imm8 (signed ext.)	83h		11-010-xxx	short	alux
ADC mem16/32, imm8 (signed ext.)	83h		mm-010-xxx	long	load, alux, store
ADD mreg8, reg8	00h		11-xxx-xxx	short	alux
ADD mem8, reg8	00h		mm-xxx-xxx	long	load, alux, store
ADD mreg16/32, reg16/32	01h		11-xxx-xxx	short	alu
ADD mem16/32, reg16/32	01h		mm-xxx-xxx	long	load, alu, store
ADD reg8, mreg8	02h		11-xxx-xxx	short	alux
ADD reg8, mem8	02h		mm-xxx-xxx	short	load, alux
ADD reg16/32, mreg16/32	03h		11-xxx-xxx	short	alu
ADD reg16/32, mem16/32	03h		mm-xxx-xxx	short	load, alu
ADD AL, imm8	04h		xx-xxx-xxx	short	alux
ADD EAX, imm16/32	05h		xx-xxx-xxx	short	alu
ADD mreg8, imm8	80h		11-000-xxx	short	alux
ADD mem8, imm8	80h		mm-000-xxx	long	load, alux, store
ADD mreg16/32, imm16/32	81h		11-000-xxx	short	alu
ADD mem16/32, imm16/32	81h		mm-000-xxx	long	load, alu, store
ADD mreg16/32, imm8 (signed ext.)	83h		11-000-xxx	short	alux
ADD mem16/32, imm8 (signed ext.)	83h		mm-000-xxx	long	load, alux, store
AND mreg8, reg8	20h		11-xxx-xxx	short	alux
AND mem8, reg8	20h		mm-xxx-xxx	long	load, alux, store
AND mreg16/32, reg16/32	21h		11-xxx-xxx	short	alu
AND mem16/32, reg16/32	21h		mm-xxx-xxx	long	load, alu, store
AND reg8, mreg8	22h		11-xxx-xxx	short	alux
AND reg8, mem8	22h		mm-xxx-xxx	short	load, alux
AND reg16/32, mreg16/32	23h		11-xxx-xxx	short	alu
AND reg16/32, mem16/32	23h		mm-xxx-xxx	short	load, alu
AND AL, imm8	24h		xx-xxx-xxx	short	alux

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
AND EAX, imm16/32	25h		xx-xxx-xxx	short	alu
AND mreg8, imm8	80h		11-100-xxx	short	alux
AND mem8, imm8	80h		mm-100-xxx	long	load, alux, store
AND mreg16/32, imm16/32	81h		11-100-xxx	short	alu
AND mem16/32, imm16/32	81h		mm-100-xxx	long	load, alu, store
AND mreg16/32, imm8 (signed ext.)	83h		11-100-xxx	short	alux
AND mem16/32, imm8 (signed ext.)	83h		mm-100-xxx	long	load, alux, store
ARPL mreg16, reg16	63h		11-xxx-xxx	vector	
ARPL mem16, reg16	63h		mm-xxx-xxx	vector	
BOUND	62h		xx-xxx-xxx	vector	
BSF reg16/32, mreg16/32	0Fh	BCh	11-xxx-xxx	vector	
BSF reg16/32, mem16/32	0Fh	BCh	mm-xxx-xxx	vector	
BSR reg16/32, mreg16/32	0Fh	BDh	11-xxx-xxx	vector	
BSR reg16/32, mem16/32	0Fh	BDh	mm-xxx-xxx	vector	
BSWAP EAX	0Fh	C8h		long	alu
BSWAP ECX	0Fh	C9h		long	alu
BSWAP EDX	0Fh	CAh		long	alu
BSWAP EBX	0Fh	CBh		long	alu
BSWAP ESP	0Fh	CCh		long	alu
BSWAP EBP	0Fh	CDh		long	alu
BSWAP ESI	0Fh	CEh		long	alu
BSWAP EDI	0Fh	CFh		long	alu
BT mreg16/32, reg16/32	0Fh	A3h	11-xxx-xxx	vector	
BT mem16/32, reg16/32	0Fh	A3h	mm-xxx-xxx	vector	
BT mreg16/32, imm8	0Fh	BAh	11-100-xxx	vector	
BT mem16/32, imm8	0Fh	BAh	mm-100-xxx	vector	
BTC mreg16/32, reg16/32	0Fh	BBh	11-xxx-xxx	vector	
BTC mem16/32, reg16/32	0Fh	BBh	mm-xxx-xxx	vector	
BTC mreg16/32, imm8	0Fh	BAh	11-111-xxx	vector	
BTC mem16/32, imm8	0Fh	BAh	mm-111-xxx	vector	
BTR mreg16/32, reg16/32	0Fh	B3h	11-xxx-xxx	vector	
BTR mem16/32, reg16/32	0Fh	B3h	mm-xxx-xxx	vector	
BTR mreg16/32, imm8	0Fh	BAh	11-110-xxx	vector	

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
BTR mem16/32, imm8	0Fh	BAh	mm-110-xxx	vector	
BTS mreg16/32, reg16/32	0Fh	ABh	11-xxx-xxx	vector	
BTS mem16/32, reg16/32	0Fh	ABh	mm-xxx-xxx	vector	
BTS mreg16/32, imm8	0Fh	BAh	11-101-xxx	vector	
BTS mem16/32, imm8	0Fh	BAh	mm-101-xxx	vector	
CALL full pointer	9Ah			vector	
CALL near imm16/32	E8h			short	store
CALL mem16:16/32	FFh		11-011-xxx	vector	
CALL near mreg32 (indirect)	FFh		11-010-xxx	vector	
CALL near mem32 (indirect)	FFh		mm-010-xxx	vector	
CBW/CWDE EAX	98h			vector	
CLC	F8h			vector	
CLD	FC			vector	
CLI	FAh			vector	
CLTS	0Fh	06h		vector	
CMC	F5h			vector	
CMP mreg8, reg8	38h		11-xxx-xxx	short	alux
CMP mem8, reg8	38h		mm-xxx-xxx	short	load, alux
CMP mreg16/32, reg16/32	39h		11-xxx-xxx	short	alu
CMP mem16/32, reg16/32	39h		mm-xxx-xxx	short	load, alu
CMP reg8, mreg8	3Ah		11-xxx-xxx	short	alux
CMP reg8, mem8	3Ah		mm-xxx-xxx	short	load, alux
CMP reg16/32, mreg16/32	3Bh		11-xxx-xxx	short	alu
CMP reg16/32, mem16/32	3Bh		mm-xxx-xxx	short	load, alu
CMP AL, imm8	3Ch		xx-xxx-xxx	short	alux
CMP EAX, imm16/32	3Dh		xx-xxx-xxx	short	alu
CMP mreg8, imm8	80h		11-111-xxx	short	alux
CMP mem8, imm8	80h		mm-111-xxx	short	load, alux
CMP mreg16/32, imm16/32	81h		11-111-xxx	short	alu
CMP mem16/32, imm16/32	81h		mm-111-xxx	short	load, alu
CMP mreg16/32, imm8 (signed ext.)	83h		11-111-xxx	long	load, alu
CMP mem16/32, imm8 (signed ext.)	83h		mm-111-xxx	long	load, alu
CMPSB mem8, mem8	A6h			vector	

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
CMP\$W mem16, mem32	A7h			vector	
CMP\$D mem32, mem32	A7h			vector	
CMPXCHG mreg8, reg8	0Fh	B0h	11-xxx-xxx	vector	
CMPXCHG mem8, reg8	0Fh	B0h	mm-xxx-xxx	vector	
CMPXCHG mreg16/32, reg16/32	0Fh	B1h	11-xxx-xxx	vector	
CMPXCHG mem16/32, reg16/32	0Fh	B1h	mm-xxx-xxx	vector	
CMPXCH8B EDX:EAX	0Fh	C7h	11-xxx-xxx	vector	
CMPXCH8B mem64	0Fh	C7h	mm-xxx-xxx	vector	
CPUID	0Fh	A2h		vector	
CWD/CDQ EDX, EAX	99h			vector	
DAA	27h			vector	
DAS	2Fh			vector	
DEC EAX	48h			short	alu
DEC ECX	49h			short	alu
DEC EDX	4Ah			short	alu
DEC EBX	4Bh			short	alu
DEC ESP	4Ch			short	alu
DEC EBP	4Dh			short	alu
DEC ESI	4Eh			short	alu
DEC EDI	4Fh			short	alu
DEC mreg8	FEh		11-001-xxx	vector	
DEC mem8	FEh		mm-001-xxx	long	load, alu, store
DEC mreg16/32	FFh		11-001-xxx	vector	
DEC mem16/32	FFh		mm-001-xxx	long	load, alu, store
DIV AL, mreg8	F6h		11-110-xxx	vector	
DIV AL, mem8	F6h		mm-110-xxx	vector	
DIV EAX, mreg16/32	F7h		11-110-xxx	vector	
DIV EAX, mem16/32	F7h		mm-110-xxx	vector	
IDIV mreg8	F6h		11-111-xxx	vector	
IDIV mem8	F6h		mm-111-xxx	vector	
IDIV EAX, mreg16/32	F7h		11-111-xxx	vector	
IDIV EAX, mem16/32	F7h		mm-111-xxx	vector	
IMUL reg16/32, imm16/32	69h		11-xxx-xxx	vector	

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
IMUL reg16/32, mreg16/32, imm16/32	69h		11-xxx-xxx	vector	
IMUL reg16/32, mem16/32, imm16/32	69h		mm-xxx-xxx	vector	
IMUL reg16/32, imm8 (sign extended)	6Bh		11-xxx-xxx	vector	
IMUL reg16/32, mreg16/32, imm8 (signed)	6Bh		11-xxx-xxx	vector	
IMUL reg16/32, mem16/32, imm8 (signed)	6Bh		mm-xxx-xxx	vector	
IMUL AX, AL, mreg8	F6h		11-101-xxx	vector	
IMUL AX, AL, mem8	F6h		mm-101-xxx	vector	
IMUL EDX:EAX, EAX, mreg16/32	F7h		11-101-xxx	vector	
IMUL EDX:EAX, EAX, mem16/32	F7h		mm-101-xxx	vector	
IMUL reg16/32, mreg16/32	0Fh	AFh	11-xxx-xxx	vector	
IMUL reg16/32, mem16/32	0Fh	AFh	mm-xxx-xxx	vector	
INC EAX	40h			short	alu
INC ECX	41h			short	alu
INC EDX	42h			short	alu
INC EBX	43h			short	alu
INC ESP	44h			short	alu
INC EBP	45h			short	alu
INC ESI	46h			short	alu
INC EDI	47h			short	alu
INC mreg8	FEh		11-000-xxx	vector	
INC mem8	FEh		mm-000-xxx	long	load, alux, store
INC mreg16/32	FFh		11-000-xxx	vector	
INC mem16/32	FFh		mm-000-xxx	long	load, alu, store
INVD	0Fh	08h		vector	
INVLPG	0Fh	01h	mm-111-xxx	vector	
JO short disp8	70h			short	branch
JB/JNAE short disp8	71h			short	branch
JNO short disp8	71h			short	branch
JNB/JAE short disp8	73h			short	branch
JZ/JE short disp8	74h			short	branch
JNZ/JNE short disp8	75h			short	branch
JBE/JNA short disp8	76h			short	branch

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
JNBE/JA short disp8	77h			short	branch
JS short disp8	78h			short	branch
JNS short disp8	79h			short	branch
JP/JPE short disp8	7Ah			short	branch
JNP/JPO short disp8	7Bh			short	branch
JL/JNGE short disp8	7Ch			short	branch
JNL/JGE short disp8	7Dh			short	branch
JLE/JNG short disp8	7Eh			short	branch
JNLE/JG short disp8	7Fh			short	branch
JXZ/JEC short disp8	E3h			vector	
JO near disp16/32	0Fh	80h		short	branch
JNO near disp16/32	0Fh	81h		short	branch
JB/JNAE near disp16/32	0Fh	82h		short	branch
JNB/JAE near disp16/32	0Fh	83h		short	branch
JZ/JE near disp16/32	0Fh	84h		short	branch
JNZ/JNE near disp16/32	0Fh	85h		short	branch
JBE/JNA near disp16/32	0Fh	86h		short	branch
JNBE/JA near disp16/32	0Fh	87h		short	branch
JS near disp16/32	0Fh	88h		short	branch
JNS near disp16/32	0Fh	89h		short	branch
JP/JPE near disp16/32	0Fh	8Ah		short	branch
JNP/JPO near disp16/32	0Fh	8Bh		short	branch
JL/JNGE near disp16/32	0Fh	8Ch		short	branch
JNL/JGE near disp16/32	0Fh	8Dh		short	branch
JLE/JNG near disp16/32	0Fh	8Eh		short	branch
JNLE/JG near disp16/32	0Fh	8Fh		short	branch
JMP near disp16/32 (direct)	E9h			short	branch
JMP far disp32/48 (direct)	EAh			vector	
JMP disp8 (short)	EBh			short	branch
JMP far mreg32 (indirect)	EFh		11-101-xxx	vector	
JMP far mem32 (indirect)	EFh		mm-101-xxx	vector	
JMP near mreg16/32 (indirect)	FFh		11-100-xxx	vector	
JMP near mem16/32 (indirect)	FFh		mm-100-xxx	vector	



# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
LAHF	9Fh			vector	
LAR reg16/32, mreg16/32	0Fh	02h	11-xxx-xxx	vector	
LAR reg16/32, mem16/32	0Fh	02h	mm-xxx-xxx	vector	
LDS reg16/32, mem32/48	C5h		mm-xxx-xxx	vector	
LEA reg16/32, mem16/32	8Dh		mm-xxx-xxx	short	load, alu
LEAVE	C9h			long	load, alu, alu
LES reg16/32, mem32/48	C4h		mm-xxx-xxx	vector	
LFS reg16/32, mem32/48	0Fh	84h		vector	
LGDT mem48	0Fh	01h	mm-010-xxx	vector	
LGS reg16/32, mem32/48	0Fh	B5h		vector	
LIDT mem48	0Fh	01h	mm-011-xxx	vector	
LLDT mreg16	0Fh	00h	11-010-xxx	vector	
LLDT mem16	0Fh	00h	mm-010-xxx	vector	
LMSW mreg16	0Fh	01h	11-100-xxx	vector	
LMSW mem16	0Fh	01h	mm-100-xxx	vector	
LODSB AL, mem8	AC			long	load, alux
LODSW AX, mem16	AD			long	load, alu
LODSD EAX, mem32	AD			long	load, alu
LOOP disp8	E2h			short	alu, branch
LOOPE/LOOPZ disp8	E1h			vector	
LOOPNE/LOOPNZ disp8	E0h			vector	
LSL reg16/32, mreg16/32	0Fh	03h	11-xxx-xxx	vector	
LSL reg16/32, mem16/32	0Fh	03h	mm-xxx-xxx	vector	
LSS reg16/32, mem32/48	0Fh	B2h	mm-xxx-xxx	vector	
LTR mreg16	0Fh	00h	11-011-xxx	vector	
LTR mem16	0Fh	00h	mm-011-xxx	vector	
MOV mreg8, reg8	88h		11-xxx-xxx	short	alux
MOV mem8, reg8	88h		mm-xxx-xxx	short	store
MOV mreg16/32, reg16/32	89h		11-xxx-xxx	short	alu
MOV mem16/32, reg16/32	89h		mm-xxx-xxx	short	store
MOV reg8, mreg8	8Ah		11-xxx-xxx	short	alux
MOV reg8, mem8	8Ah		mm-xxx-xxx	short	load
MOV reg16/32, mreg16/32	8Bh		11-xxx-xxx	short	alu

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
MOV reg16/32, mem16/32	8Bh		mm-xxx-xxx	short	load
MOV mreg16, segment reg	8Ch		11-xxx-xxx	long	load
MOV mem16, segment reg	8Ch		mm-xxx-xxx	vector	
MOV segment reg, mreg16	8Eh		11-xxx-xxx	vector	
MOV segment reg, mem16	8Eh		mm-xxx-xxx	vector	
MOV AL, mem8	A0h			short	load
MOV EAX, mem16/32	A1h			short	load
MOV mem8, AL	A2h			short	store
MOV mem16/32, EAX	A3h			short	store
MOV AL, imm8	B0h			short	limm
MOV CL, imm8	B1h			short	limm
MOV DL, imm8	B2h			short	limm
MOV BL, imm8	B3h			short	limm
MOV AH, imm8	B4h			short	limm
MOV CH, imm8	B5h			short	limm
MOV DH, imm8	B6h			short	limm
MOV BH, imm8	B7h			short	limm
MOV EAX, imm16/32	B8h			short	limm
MOV ECX, imm16/32	B9h			short	limm
MOV EDX, imm16/32	BAh			short	limm
MOV EBX, imm16/32	BBh			short	limm
MOV ESP, imm16/32	BCh			short	limm
MOV EBP, imm16/32	BDh			short	limm
MOV ESI, imm16/32	BEh			short	limm
MOV EDI, imm16/32	BFh			short	limm
MOV mreg8, imm8	C6h		11-000-xxx	short	limm
MOV mem8, imm8	C6h		mm-000-xxx	long	store
MOV reg16/32, imm16/32	C7h		11-000-xxx	short	limm
MOV mem16/32, imm16/32	C7h		mm-000-xxx	long	store
MOVS8 mem8, mem8	A4h			long	load, store, alux, alux
MOVSD mem16, mem16	A5h			long	load, store, alu, alu
MOVSW mem32, mem32	A5h			long	load, store, alu, alu
MOVX reg16/32, mreg8	0Fh	BEh	11-xxx-xxx	short	alu

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
MOVX reg16/32, mem8	0Fh	BEh	mm-xxx-xxx	short	load, alu
MOVX reg32, mreg16	0Fh	BFh	11-xxx-xxx	short	alu
MOVX reg32, mem16	0Fh	BFh	mm-xxx-xxx	short	load, alu
MOVZX reg16/32, mreg8	0Fh	B6h	11-xxx-xxx	short	alu
MOVZX reg16/32, mem8	0Fh	B6h	mm-xxx-xxx	short	load, alu
MOVZX reg32, mreg16	0Fh	B7h	11-xxx-xxx	short	alu
MOVZX reg32, mem16	0Fh	B7h	mm-xxx-xxx	short	load, alu
MUL AL, mreg8	F6h		11-100-xxx	vector	
MUL AL, mem8	F6h		mm-100-xxx	vector	
MUL EAX, mreg16/32	F7h		11-100-xxx	vector	
MUL EAX, mem16/32	F7h		mm-100-xxx	vector	
NEG mreg8	F6h		11-011-xxx	short	alux
NEG mem8	F6h		mm-011-xxx	vector	
NEG mreg16/32	F7h		11-011-xxx	short	alu
NEG mem16/32	F7h		mm-011-xxx	vector	
NOP (XCHG AX, AX)	90h			short	limm
NOT mreg8	F6h		11-010-xxx	short	alux
NOT mem8	F6h		mm-010-xxx	vector	
NOT mreg16/32	F7h		11-010-xxx	short	alu
NOT mem16/32	F7h		mm-010-xxx	vector	
OR mreg8, reg8	08h		11-xxx-xxx	short	alux
OR mem8, reg8	08h		mm-xxx-xxx	long	load, alux, store
OR mreg16/32, reg16/32	09h		11-xxx-xxx	short	alu
OR mem16/32, reg16/32	09h		mm-xxx-xxx	long	load, alu, store
OR reg8, mreg8	0Ah		11-xxx-xxx	short	alux
OR reg8, mem8	0Ah		mm-xxx-xxx	short	load, alux
OR reg16/32, mreg16/32	0Bh		11-xxx-xxx	short	alu
OR reg16/32, mem16/32	0Bh		mm-xxx-xxx	short	load, alu
OR AL, imm8	0Ch		xx-xxx-xxx	short	alux
OR EAX, imm16/32	0Dh		xx-xxx-xxx	short	alu
OR mreg8, imm8	80h		11-001-xxx	short	alux
OR mem8, imm8	80h		mm-001-xxx	long	load, alux, store
OR mreg16/32, imm16/32	81h		11-001-xxx	short	alu

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
OR mem16/32, imm16/32	81h		mm-001-xxx	long	load, alu, store
OR mreg16/32, imm8 (signed ext.)	83h		11-001-xxx	short	alux
OR mem16/32, imm8 (signed ext.)	83h		mm-001-xxx	long	load, alux, store
POP ES	07h			vector	
POP SS	17h			vector	
POP DS	1Fh			vector	
POP FS	0Fh	A1h		vector	
POP GS	0Fh	A9h		vector	
POP EAX	58h			short	load, alu
POP ECX	59h			short	load, alu
POP EDX	5Ah			short	load, alu
POP EBX	5Bh			short	load, alu
POP ESP	5Ch			short	load, alu
POP EBP	5Dh			short	load, alu
POP ESI	5Eh			short	load, alu
POP EDI	5Fh			short	load, alu
POP mreg	8Fh		11-000-xxx	short	load, alu
POP mem	8Fh		mm-000-xxx	long	load, store, alu
POPA/POPAD	61h			vector	
POPF/POPFD	9Dh			vector	
PUSH ES	06h			long	load, store
PUSH CS	0Eh			vector	
PUSH FS	0Fh	A0h		vector	
PUSH GS	0Fh	A8h		vector	
PUSH SS	16h			vector	
PUSH DS	1Eh			long	load, store
PUSH EAX	50h			short	store
PUSH ECX	51h			short	store
PUSH EDX	52h			short	store
PUSH EBX	53h			short	store
PUSH ESP	54h			short	store
PUSH EBP	55h			short	store
PUSH ESI	56h			short	store

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC66 Opcodes
PUSH EDI	57h			short	store
PUSH imm8	6Ah			long	store
PUSH imm16/32	68h			long	store
PUSH mreg16/32	FFh		11-110-xxx	vector	
PUSH mem16/32	FFh		mm-110-xxx	long	load, store
PUSHA/PUSHAD	60h			vector	
PUSHF/PUSHFD	9Ch			vector	
RCL mreg8, imm8	C0h		11-010-xxx	vector	
RCL mem8, imm8	C0h		mm-010-xxx	vector	
RCL mreg16/32, imm8	C1h		11-010-xxx	vector	
RCL mem16/32, imm8	C1h		mm-010-xxx	vector	
RCL mreg8, 1	D0h		11-010-xxx	vector	
RCL mem8, 1	D0h		mm-010-xxx	vector	
RCL mreg16/32, 1	D1h		11-010-xxx	vector	
RCL mem16/32, 1	D1h		mm-010-xxx	vector	
RCL mreg8, CL	D2h		11-010-xxx	vector	
RCL mem8, CL	D2h		mm-010-xxx	vector	
RCL mreg16/32, CL	D3h		11-010-xxx	vector	
RCL mem16/32, CL	D3h		mm-010-xxx	vector	
RCR mreg8, imm8	C0h		11-011-xxx	vector	
RCR mem8, imm8	C0h		mm-011-xxx	vector	
RCR mreg16/32, imm8	C1h		11-011-xxx	vector	
RCR mem16/32, imm8	C1h		mm-011-xxx	vector	
RCR mreg8, 1	D0h		11-011-xxx	vector	
RCR mem8, 1	D0h		mm-011-xxx	vector	
RCR mreg16/32, 1	D1h		11-011-xxx	vector	
RCR mem16/32, 1	D1h		mm-011-xxx	vector	
RCR mreg8, CL	D2h		11-011-xxx	vector	
RCR mem8, CL	D2h		mm-011-xxx	vector	
RCR mreg16/32, CL	D3h		11-011-xxx	vector	
RCR mem16/32, CL	D3h		mm-011-xxx	vector	
RET near imm16	C2h			vector	
RET near	C3h			vector	

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
RET far imm16	CAh			vector	
RET far	CBh			vector	
ROL mreg8, imm8	C0h		11-000-xxx	vector	
ROL mem8, imm8	C0h		mm-000-xxx	vector	
ROL mreg16/32, imm8	C1h		11-000-xxx	vector	
ROL mem16/32, imm8	C1h		mm-000-xxx	vector	
ROL mreg8, 1	D0h		11-000-xxx	vector	
ROL mem8, 1	D0h		mm-000-xxx	vector	
ROL mreg16/32, 1	D1h		11-000-xxx	vector	
ROL mem16/32, 1	D1h		mm-000-xxx	vector	
ROL mreg8, CL	D2h		11-000-xxx	vector	
ROL mem8, CL	D2h		mm-000-xxx	vector	
ROL mreg16/32, CL	D3h		11-000-xxx	vector	
ROL mem16/32, CL	D3h		mm-000-xxx	vector	
ROR mreg8, imm8	C0h		11-001-xxx	vector	
ROR mem8, imm8	C0h		mm-001-xxx	vector	
ROR mreg16/32, imm8	C1h		11-001-xxx	vector	
ROR mem16/32, imm8	C1h		mm-001-xxx	vector	
ROR mreg8, 1	D0h		11-001-xxx	vector	
ROR mem8, 1	D0h		mm-001-xxx	vector	
ROR mreg16/32, 1	D1h		11-001-xxx	vector	
ROR mem16/32, 1	D1h		mm-001-xxx	vector	
ROR mreg8, CL	D2h		11-001-xxx	vector	
ROR mem8, CL	D2h		mm-001-xxx	vector	
ROR mreg16/32, CL	D3h		11-001-xxx	vector	
ROR mem16/32, CL	D3h		mm-001-xxx	vector	
SAHF	9Eh			vector	
SAR mreg8, imm8	C0h		11-111-xxx	short	alux
SAR mem8, imm8	C0h		mm-111-xxx	vector	
SAR mreg16/32, imm8	C1h		11-111-xxx	short	alu
SAR mem16/32, imm8	C1h		mm-111-xxx	vector	
SAR mreg8, 1	D0h		11-111-xxx	short	alux
SAR mem8, 1	D0h		mm-111-xxx	vector	

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
SAR mreg16/32, I	D1h		11-111-xxx	short	alu
SAR mem16/32, I	D1h		mm-111-xxx	vector	
SAR mreg8, CL	D2h		11-111-xxx	short	alux
SAR mem8, CL	D2h		mm-111-xxx	vector	
SAR mreg16/32, CL	D3h		11-111-xxx	short	alu
SAR mem16/32, CL	D3h		mm-111-xxx	vector	
SBB mreg8, reg8	18h		11-xxx-xxx	short	alux
SBB mem8, reg8	18h		mm-xxx-xxx	long	load, alux, store
SBB mreg16/32, reg16/32	19h		11-xxx-xxx	short	alu
SBB mem16/32, reg16/32	19h		mm-xxx-xxx	long	load, alu, store
SBB reg8, mreg8	1Ah		11-xxx-xxx	short	alux
SBB reg8, mem8	1Ah		mm-xxx-xxx	short	load, alux
SBB reg16/32, mreg16/32	1Bh		11-xxx-xxx	short	alu
SBB reg16/32, mem16/32	1Bh		mm-xxx-xxx	short	load, alu
SBB AL, imm8	1Ch		xx-xxx-xxx	short	alux
SBB EAX, imm16/32	1Dh		xx-xxx-xxx	short	alu
SBB mreg8, imm8	80h		11-011-xxx	short	alux
SBB mem8, imm8	80h		mm-011-xxx	long	load, alux, store
SBB mreg16/32, imm16/32	81h		11-011-xxx	short	alu
SBB mem16/32, imm16/32	81h		mm-011-xxx	long	load, alu, store
SBB mreg8, imm8 (signed ext.)	83h		11-011-xxx	short	alux
SBB mem8, imm8 (signed ext.)	83h		mm-011-xxx	long	load, alux, store
SCASB AL, mem8	A Eh			vector	
SCASW AX, mem16	A Fh			vector	
SCASD EAX, mem32	A Fh			vector	
SETO mreg8	0Fh	90h	11-xxx-xxx	vector	
SETO mem8	0Fh	90h	mm-xxx-xxx	vector	
SETNO mreg8	0Fh	91h	11-xxx-xxx	vector	
SETNO mem8	0Fh	91h	mm-xxx-xxx	vector	
SETB/SETNAE mreg8	0Fh	92h	11-xxx-xxx	vector	
SETB/SETNAE mem8	0Fh	92h	mm-xxx-xxx	vector	
SETNB/SETAE mreg8	0Fh	93h	11-xxx-xxx	vector	
SETNB/SETAE mem8	0Fh	93h	mm-xxx-xxx	vector	

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
SETZ/SETE mreg8	0Fh	94h	11-xxx-xxx	vector	
SETZ/SETE mem8	0Fh	94h	mm-xxx-xxx	vector	
SETNZ/SETNE mreg8	0Fh	95h	11-xxx-xxx	vector	
SETNZ/SETNE mem8	0Fh	95h	mm-xxx-xxx	vector	
SETBE/SETNA mreg8	0Fh	96h	11-xxx-xxx	vector	
SETBE/SETNA mem8	0Fh	96h	mm-xxx-xxx	vector	
SETNBE/SETA mreg8	0Fh	97h	11-xxx-xxx	vector	
SETNBE/SETA mem8	0Fh	97h	mm-xxx-xxx	vector	
SETS mreg8	0Fh	98h	11-xxx-xxx	vector	
SETS mem8	0Fh	98h	mm-xxx-xxx	vector	
SETNS mreg8	0Fh	99h	11-xxx-xxx	vector	
SETNS mem8	0Fh	99h	mm-xxx-xxx	vector	
SETP/SETPE mreg8	0Fh	9Ah	11-xxx-xxx	vector	
SETP/SETPE mem8	0Fh	9Ah	mm-xxx-xxx	vector	
SETNP/SETPO mreg8	0Fh	9Bh	11-xxx-xxx	vector	
SETNP/SETPO mem8	0Fh	9Bh	mm-xxx-xxx	vector	
SETL/SETNGE mreg8	0Fh	9Ch	11-xxx-xxx	vector	
SETL/SETNGE mem8	0Fh	9Ch	mm-xxx-xxx	vector	
SETNL/SETGE mreg8	0Fh	9Dh	11-xxx-xxx	vector	
SETNL/SETGE mem8	0Fh	9Dh	mm-xxx-xxx	vector	
SETLE/SETNG mreg8	0Fh	9Eh	11-xxx-xxx	vector	
SETLE/SETNG mem8	0Fh	9Eh	mm-xxx-xxx	vector	
SETNLE/SETG mreg8	0Fh	9Fh	11-xxx-xxx	vector	
SETNLE/SETG mem8	0Fh	9Fh	mm-xxx-xxx	vector	
SGDT mem48	0Fh	01h	mm-000-xxx	vector	
SIDT mem48	0Fh	01h	mm-001-xxx	vector	
SHL/SAL mreg8, imm8	C0h		11-100-xxx	short	alux
SHL/SAL mem8, imm8	C0h		mm-100-xxx	vector	
SHL/SAL mreg16/32, imm8	C1h		11-100-xxx	short	alu
SHL/SAL mem16/32, imm8	C1h		mm-100-xxx	vector	
SHL/SAL mreg8, 1	D0h		11-100-xxx	short	alux
SHL/SAL mem8, 1	D0h		mm-100-xxx	vector	
SHL/SAL mreg16/32, 1	D1h		11-100-xxx	short	alu



# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
SHL/SAL mem16/32, 1	D1h		mm-100-xxx	vector	
SHL/SAL mreg8, CL	D2h		11-100-xxx	short	alux
SHL/SAL mem8, CL	D2h		mm-100-xxx	vector	
SHL/SAL mreg16/32, CL	D3h		11-100-xxx	short	alu
SHL/SAL mem16/32, CL	D3h		mm-100-xxx	vector	
SHR mreg8, imm8	C0h		11-101-xxx	short	alux
SHR mem8, imm8	C0h		mm-101-xxx	vector	
SHR mreg16/32, imm8	C1h		11-101-xxx	short	alu
SHR mem16/32, imm8	C1h		mm-101-xxx	vector	
SHR mreg8, 1	D0h		11-101-xxx	short	alux
SHR mem8, 1	D0h		mm-101-xxx	vector	
SHR mreg16/32, 1	D1h		11-101-xxx	short	alu
SHR mem16/32, 1	D1h		mm-101-xxx	vector	
SHR mreg8, CL	D2h		11-101-xxx	short	alux
SHR mem8, CL	D2h		mm-101-xxx	vector	
SHR mreg16/32, CL	D3h		11-101-xxx	short	alu
SHR mem16/32, CL	D3h		mm-101-xxx	vector	
SHLD mreg16/32, reg16/32, imm8	0Fh	A4h	11-xxx-xxx	vector	
SHLD mem16/32, reg16/32, imm8	0Fh	A4h	mm-xxx-xxx	vector	
SHLD mreg16/32, reg16/32, CL	0Fh	A5h	11-xxx-xxx	vector	
SHLD mem16/32, reg16/32, CL	0Fh	A5h	mm-xxx-xxx	vector	
SHRD mreg16/32, reg16/32, imm8	0Fh	ACH	11-xxx-xxx	vector	
SHRD mem16/32, reg16/32, imm8	0Fh	ACH	mm-xxx-xxx	vector	
SHRD mreg16/32, reg16/32, CL	0Fh	ADh	11-xxx-xxx	vector	
SHRD mem16/32, reg16/32, CL	0Fh	ADh	mm-xxx-xxx	vector	
SLDT mreg16	0Fh	00h	11-000-xxx	vector	
SLDT mem16	0Fh	00h	mm-000-xxx	vector	
SMSW mreg16	0Fh	01h	11-100-xxx	vector	
SMSW mem16	0Fh	01h	mm-100-xxx	vector	
STC	F9h			vector	
STD	FDh			vector	
STI	FBh			vector	
STOSB mem8, AL	AAh			long	store, alux

**Table 12. Integer Instructions (continued)**

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
STOSW mem16, AX	ABh			long	store, alu
STOSD mem32, EAX	ABh			long	store, alu
STR mreg16	0Fh	00h	11-001-xxx	vector	
STR mem16	0Fh	00h	mm-001-xxx	vector	
SUB mreg8, reg8	28h		11-xxx-xxx	short	alux
SUB mem8, reg8	28h		mm-xxx-xxx	long	load, alux, store
SUB mreg16/32, reg16/32	29h		11-xxx-xxx	short	alu
SUB mem16/32, reg16/32	29h		mm-xxx-xxx	long	load, alu, store
SUB reg8, mreg8	2Ah		11-xxx-xxx	short	alux
SUB reg8, mem8	2Ah		mm-xxx-xxx	short	load, alux
SUB reg16/32, mreg16/32	2Bh		11-xxx-xxx	short	alu
SUB reg16/32, mem16/32	2Bh		mm-xxx-xxx	short	load, alu
SUB AL, imm8	2Ch		xx-xxx-xxx	short	alux
SUB EAX, imm16/32	2Dh		xx-xxx-xxx	short	alu
SUB mreg8, imm8	80h		11-101-xxx	short	alux
SUB mem8, imm8	80h		mm-101-xxx	long	load, alux, store
SUB mreg16/32, imm16/32	81h		11-101-xxx	short	alu
SUB mem16/32, imm16/32	81h		mm-101-xxx	long	load, alu, store
SUB mreg16/32, imm8 (signed ext.)	83h		11-101-xxx	short	alux
SUB mem16/32, imm8 (signed ext.)	83h		mm-101-xxx	long	load, alux, store
SYSCALL	0Fh	05h		vector	
SYSRET	0Fh	07h		vector	
TEST mreg8, reg8	84h		11-xxx-xxx	short	alux
TEST mem8, reg8	84h		mm-xxx-xxx	vector	
TEST mreg16/32, reg16/32	85h		11-xxx-xxx	short	alu
TEST mem16/32, reg16/32	85h		mm-xxx-xxx	vector	
TEST AL, imm8	A8h			long	alux
TEST EAX, Imm16/32	A9h			long	alu
TEST mreg8, imm8	F6h		11-000-xxx	long	alux
TEST mem8, imm8	F6h		mm-000-xxx	long	load, alux
TEST mreg16/32, imm16/32	F7h		11-000-xxx	long	alu
TEST mem16/32, imm16/32	F7h		mm-000-xxx	long	load, alu
VERR mreg16	0Fh	00h	11-100-xxx	vector	

# 3 Software Environment

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
VERR mem16	0Fh	00h	mm-100-xxx	vector	
VERW mreg16	0Fh	00h	11-101-xxx	vector	
VERW mem16	0Fh	00h	mm-101-xxx	vector	
WAIT	9Bh			vector	
WBINVD	0Fh	09h		vector	
XADD mreg8, reg8	0Fh	C0h	11-100-xxx	vector	
XADD mem8, reg8	0Fh	C0h	mm-100-xxx	vector	
XADD mreg16/32, reg16/32	0Fh	C1h	11-101-xxx	vector	
XADD mem16/32, reg16/32	0Fh	C1h	mm-101-xxx	vector	
XCHG reg8, mreg8	86h		11-xxx-xxx	vector	
XCHG reg8, mem8	86h		mm-xxx-xxx	vector	
XCHG reg16/32, mreg16/32	87h		11-xxx-xxx	vector	
XCHG reg16/32, mem16/32	87h		mm-xxx-xxx	vector	
XCHG EAX, EAX	90h			short	limm
XCHG EAX, ECX	91h			long	alu, alu, alu
XCHG EAX, EDX	92h			long	alu, alu, alu
XCHG EAX, EBX	93h			long	alu, alu, alu
XCHG EAX, ESP	94h			long	alu, alu, alu
XCHG EAX, EBP	95h			long	alu, alu, alu
XCHG EAX, ESI	96h			long	alu, alu, alu
XCHG EAX, EDI	97h			long	alu, alu, alu
XLAT	D7h			vector	
XOR mreg8, reg8	30h		11-xxx-xxx	short	alux
XOR mem8, reg8	30h		mm-xxx-xxx	long	load, alux, store
XOR mreg16/32, reg16/32	31h		11-xxx-xxx	short	alu
XOR mem16/32, reg16/32	31h		mm-xxx-xxx	long	load, alu, store
XOR reg8, mreg8	32h		11-xxx-xxx	short	alux
XOR reg8, mem8	32h		mm-xxx-xxx	short	load, alux
XOR reg16/32, mreg16/32	33h		11-xxx-xxx	short	alu
XOR reg16/32, mem16/32	33h		mm-xxx-xxx	short	load, alu
XOR AL, imm8	34h		xx-xxx-xxx	short	alux
XOR EAX, imm16/32	35h		xx-xxx-xxx	short	alu
XOR mreg8, imm8	80h		11-110-xxx	short	alux

Table 12. Integer Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes
XOR mem8, imm8	80h		mm-110-xxx	long	load, alux, store
XOR mreg16/32, imm16/32	81h		11-110-xxx	short	alu
XOR mem16/32, imm16/32	81h		mm-110-xxx	long	load, alu, store
XOR mreg16/32, imm8 (signed ext.)	83h		11-110-xxx	short	alux
XOR mem16/32, imm8 (signed ext.)	83h		mm-110-xxx	long	load, alux, store

Table 13. Floating-Point Instructions

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
F2XM1	D9h	F0h		short	float	
FABS	D9h	F1h		short	float	
FADD ST(0), ST(i)	D8h		11-000-xxx	short	float	*
FADD ST(0), mem32real	D8h		mm-000-xxx	short	fload, float	
FADD ST(i), ST(0)	DCh		11-000-xxx	short	float	*
FADD ST(0), mem64real	DCh		mm-000-xxx	short	fload, float	
FADDP ST(i), ST(0)	DEh		11-000-xxx	short	float	*
FBLD	DFh		mm-100-xxx	vector		*
FBSTP	DFh		mm-110-xxx	vector		*
FCHS	D9h	E0h		short	float	
FCLEX	DBh	E2h		vector		
FCOM ST(0), ST(i)	D8h		11-010-xxx	short	float	*
FCOM ST(0), mem32real	D8h		mm-010-xxx	short	fload, float	
FCOM ST(0), mem64real	DCh		mm-010-xxx	short	fload, float	
FCOMP ST(0), ST(i)	D8h		11-011-xxx	short	float	*
FCOMP ST(0), mem32real	D8h		mm-011-xxx	short	fload, float	
FCOMP ST(0), mem64real	DCh		mm-011-xxx	short	fload, float	
FCOMPP	DEh		11-011-001	short	float	
FCOS ST(0)	D9h	FFh		short	float	
FDECSTP	D9h	F6h		short	float	
FDIV ST(0), ST(i) (single precision)	D8h		11-110-xxx	short	float	*
FDIV ST(0), ST(i) (double precision)	D8h		11-110-xxx	short	float	*
<b>Note:</b> * The last three bits of the modR/M byte select the stack entry ST(i).						

# 3 Software Environment

**Table 13. Floating-Point Instructions (continued)**

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
FDIV ST(0), ST(i) (extended precision)	D8h		11-110-xxx	short	float	*
FDIV ST(i), ST(0) (single precision)	DCh		11-111-xxx	short	float	*
FDIV ST(i), ST(0) (double precision)	DCh		11-111-xxx	short	float	*
FDIV ST(i), ST(0) (extended precision)	DCh		11-111-xxx	short	float	*
FDIV ST(0), mem32real	D8h		mm-110-xxx	short	fload, float	
FDIV ST(0), mem64real	DCh		mm-110-xxx	short	fload, float	
FDIVP ST(0), ST(i)	DEh		11-111-xxx	short	float	*
FDIVR ST(0), ST(i)	D8h		11-110-xxx	short	float	*
FDIVR ST(i), ST(0)	DCh		11-111-xxx	short	float	*
FDIVR ST(0), mem32real	D8h		mm-111-xxx	short	fload, float	
FDIVR ST(0), mem64real	DCh		mm-111-xxx	short	fload, float	
FDIVRP ST(i), ST(0)	DEh		11-110-xxx	short	float	*
FFREE ST(i)	DDh		11-000-xxx	short	float	*
FIADD ST(0), mem32int	DAh		mm-000-xxx	short	fload, float	
FIADD ST(0), mem16int	DEh		mm-000-xxx	short	fload, float	
FICOM ST(0), mem32int	DAh		mm-010-xxx	short	fload, float	
FICOM ST(0), mem16int	DEh		mm-010-xxx	short	fload, float	
FICOMP ST(0), mem32int	DAh		mm-011-xxx	short	fload, float	
FICOMP ST(0), mem16int	DEh		mm-011-xxx	short	fload, float	
FIDIV ST(0), mem32int	DAh		mm-110-xxx	short	fload, float	
FIDIV ST(0), mem16int	DEh		mm-110-xxx	short	fload, float	
FIDIVR ST(0), mem32int	DAh		mm-111-xxx	short	fload, float	
FIDIVR ST(0), mem16int	DEh		mm-111-xxx	short	fload, float	
FILD mem16int	DFh		mm-000-xxx	short	fload, float	
FILD mem32int	DBh		mm-000-xxx	short	fload, float	
FILD mem64int	DFh		mm-101-xxx	short	fload, float	
FIMUL ST(0), mem32int	DAh		mm-001-xxx	short	fload, float	
FIMUL ST(0), mem16int	DEh		mm-001-xxx	short	fload, float	
FINCSTP	D9h	F7h		short	float	
FINIT	DBh	E3h		vector		
FIST mem16int	DFh		mm-010-xxx	short	fload, float	
<b>Note:</b> * The last three bits of the modR/M byte select the stack entry ST(i).						

Table 13. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
FIST mem32int	DBh		mm-010-xxx	short	fload, float	
FISTP mem16int	DFh		mm-011-xxx	short	fload, float	
FISTP mem32int	DBh		mm-011-xxx	short	fload, float	
FISTP mem64int	DFh		mm-111-xxx	short	fload, float	
FISUB ST(0), mem32int	DAh		mm-100-xxx	short	fload, float	
FISUB ST(0), mem16int	DEh		mm-100-xxx	short	fload, float	
FISUBR ST(0), mem32int	DAh		mm-101-xxx	short	fload, float	
FISUBR ST(0), mem16int	DEh		mm-101-xxx	short	fload, float	
FLD ST(i)	D9h		11-000-xxx	short	fload, float	*
FLD mem32real	D9h		mm-000-xxx	short	fload, float	
FLD mem64real	DDh		mm-000-xxx	short	fload, float	
FLD mem80real	DBh		mm-101-xxx	vector		
FLDI	D9h	E8h		short	fload, float	
FLDCW	D9h		mm-101-xxx	vector		
FLDENV	D9h		mm-100-xxx	short	fload, float	
FLDL2E	D9h	EAh		short	float	
FLDL2T	D9h	E9h		short	float	
FLDLG2	D9h	ECh		short	float	
FLDLN2	D9h	EDh		short	float	
FLDPI	D9h	EBh		short	float	
FLDZ	D9h	EEh		short	float	
FMUL ST(0), ST(i)	D8h		11-001-xxx	short	float	*
FMUL ST(i), ST(0)	DCh		11-001-xxx	short	float	*
FMUL ST(0), mem32real	D8h		mm-001-xxx	short	fload, float	
FMUL ST(0), mem64real	DCh		mm-001-xxx	short	fload, float	
FMULP ST(0), ST(i)	DEh		11-001-xxx	short	float	
FNOP	D9h	D0h		short	float	
FPATAN	D9h	F3h		short	float	
FPREM	D9h	F8h		short	float	
FPREM1	D9h	F5h		short	float	
FPTAN	D9h	F2h		vector		
<b>Note:</b> * The last three bits of the modR/M byte select the stack entry ST(i).						

# 3 Software Environment

**Table 13. Floating-Point Instructions (continued)**

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
FRNDINT	D9h	FCh		short	float	
FRSTOR	DDh		mm-100-xxx	vector		
FSAVE	DDh		mm-110-xxx	vector		
FSCALE	D9h	FDh		short	float	
FSIN	D9h	FEh		short	float	
FSINCOS	D9h	FBh		vector		
FSQRT (single precision)	D9h	FAh		short	float	
FSQRT (double precision)	D9h	FAh		short	float	
FSQRT (extended precision)	D9h	FAh		short	float	
FST mem32real	D9h		mm-010-xxx	short	fstore	
FST mem64real	DDh		mm-010-xxx	short	fstore	
FST ST(i)	DDh		11-010xxx	short	fstore	
FSTCW	D9h		mm-111-xxx	vector		
FSTENV	D9h		mm-110-xxx	vector		
FSTP mem32real	D9h		mm-011-xxx	short	fstore	
FSTP mem64real	DDh		mm-011-xxx	short	fstore	
FSTP mem80real	D9h		mm-111-xxx	vector		
FSTP ST(i)	DDh		11-011-xxx	short	float	
FSTSW AX	DFh	E0h		vector		
FSTSW mem16	DDh		mm-111-xxx	vector		
FSUB ST(0), mem32real	D8h		mm-100-xxx	short	fload, float	
FSUB ST(0), mem64real	DCh		mm-100-xxx	short	fload, float	
FSUB ST(0), ST(i)	D8h		11-100-xxx	short	float	
FSUB ST(i), ST(0)	DCh		11-101-xxx	short	float	
FSUBP ST(0), ST(i)	DEh		11-101-xxx	short	float	
FSUBR ST(0), mem32real	D8h		mm-101-xxx	short	fload, float	
FSUBR ST(0), mem64real	DCh		mm-101-xxx	short	fload, float	
FSUBR ST(0), ST(i)	D8h		11-100-xxx	short	float	
FSUBR ST(i), ST(0)	DCh		11-101-xxx	short	float	
FSUBRP ST(i), ST(0)	DEh		11-100-xxx	short	float	
FTST	D9h	E4h		short	float	
<b>Note:</b> * The last three bits of the modR/M byte select the stack entry ST(i).						

Table 13. Floating-Point Instructions (continued)

Instruction Mnemonic	First Byte	Second Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
FUCOM	DDh		11-100-xxx	short	float	
FUCOMP	DDh		11-101-xxx	short	float	
FUCOMPP	DAh	E9h		short	float	
FXAM	D9h	E5h		short	float	
FXCH	D9h		11-001-xxx	short	float	
FXTRACT	D9h	F4h		vector		
FYL2X	D9h	F1h		short	float	
FYL2XP1	D9h	F9h		short	float	
FWAIT	9Bh			vector		
<b>Note:</b> * The last three bits of the modR/M byte select the stack entry ST(i).						

For more information about MMX instructions, see Appendix A, “MMX Multimedia Technology” on page 347.

Table 14. MMX Instructions

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
EMMS	0Fh	77h		vector		
MOVD mmreg, mreg32	0Fh	6Eh	11-xxx-xxx	short	store, mload	**
MOVD mmreg, mem32	0Fh	6Eh	mm-xxx-xxx	short	mload	
MOVD mreg32, mmreg	0Fh	7Eh	11-xxx-xxx	short	mstore, load	**
MOVD mem32, mmreg	0Fh	7Eh	mm-xxx-xxx	short	mstore	
MOVQ mmreg1, mmreg2	0Fh	6Fh	11-xxx-xxx	short	mmx	
MOVQ mmreg, mem64	0Fh	6Fh	mm-xxx-xxx	short	mload, mload	
MOVQ mmreg1, mmreg2	0Fh	7Fh	11-xxx-xxx	short	mmx	
MOVQ mem64, mmreg	0Fh	7Fh	mm-xxx-xxx	short	mload, mstore	
PACKSSDW mmreg1, mmreg2	0Fh	6Bh	11-xxx-xxx	short	mmx	
PACKSSDW mmreg, mem64	0Fh	6Bh	mm-xxx-xxx	short	mload, mmx	
PACKSSWB mmreg1, mmreg2	0Fh	63h	11-xxx-xxx	short	mmx	
PACKSSWB mmreg, mem64	0Fh	64h	mm-xxx-xxx	short	mload, mmx	
PACKUSWB mmreg1, mmreg2	0Fh	67h	11-xxx-xxx	short	mmx	
<b>Note:</b> ** Bits 2, 1, and 0 of the modR/M byte select the integer register.						



# 3 Software Environment

Table 14. MMX Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
PACKUSWB mmreg, mem64	0Fh	67h	mm-xxx-xxx	short	mload, mmx	
PADDB mmreg1, mmreg2	0Fh	FCh	11-xxx-xxx	short	mmx	
PADDB mmreg, mem64	0Fh	FCh	mm-xxx-xxx	short	mload, mmx	
PADD mmreg1, mmreg2	0Fh	FEh	11-xxx-xxx	short	mmx	
PADD mmreg, mem64	0Fh	FEh	mm-xxx-xxx	short	mload, mmx	
PADDSB mmreg1, mmreg2	0Fh	ECh	11-xxx-xxx	short	mmx	
PADDSB mmreg, mem64	0Fh	ECh	mm-xxx-xxx	short	mload, mmx	
PADDSW mmreg1, mmreg2	0Fh	EDh	11-xxx-xxx	short	mmx	
PADDSW mmreg, mem64	0Fh	EDh	mm-xxx-xxx	short	mload, mmx	
PADDUSB mmreg1, mmreg2	0Fh	DCh	11-xxx-xxx	short	mmx	
PADDUSB mmreg, mem64	0Fh	DCh	mm-xxx-xxx	short	mload, mmx	
PADDUSW mmreg1, mmreg2	0Fh	DDh	11-xxx-xxx	short	mmx	
PADDUSW mmreg, mem64	0Fh	DDh	mm-xxx-xxx	short	mload, mmx	
PADDW mmreg1, mmreg2	0Fh	FDh	11-xxx-xxx	short	mmx	
PADDW mmreg, mem64	0Fh	FDh	mm-xxx-xxx	short	mload, mmx	
PAND mmreg1, mmreg2	0Fh	DBh	11-xxx-xxx	short	mmx	
PAND mmreg, mem64	0Fh	DBh	mm-xxx-xxx	short	mload, mmx	
PANDN mmreg1, mmreg2	0Fh	DFh	11-xxx-xxx	short	mmx	
PANDN mmreg, mem64	0Fh	DFh	mm-xxx-xxx	short	mload, mmx	
PCMPEQB mmreg1, mmreg2	0Fh	74h	11-xxx-xxx	short	mmx	
PCMPEQB mmreg, mem64	0Fh	74h	mm-xxx-xxx	short	mload, mmx	
PCMPEQD mmreg1, mmreg2	0Fh	76h	11-xxx-xxx	short	mmx	
PCMPEQD mmreg, mem64	0Fh	76h	mm-xxx-xxx	short	mload, mmx	
PCMPEQW mmreg1, mmreg2	0Fh	75h	11-xxx-xxx	short	mmx	
PCMPEQW mmreg, mem64	0Fh	75h	mm-xxx-xxx	short	mload, mmx	
PCMPGTB mmreg1, mmreg2	0Fh	64h	11-xxx-xxx	short	mmx	
PCMPGTB mmreg, mem64	0Fh	64h	mm-xxx-xxx	short	mload, mmx	
PCMPGTD mmreg1, mmreg2	0Fh	66h	11-xxx-xxx	short	mmx	
PCMPGTD mmreg, mem64	0Fh	66h	mm-xxx-xxx	short	mload, mmx	
PCMPGTW mmreg1, mmreg2	0Fh	65h	11-xxx-xxx	short	mmx	
PCMPGTW mmreg, mem64	0Fh	65h	mm-xxx-xxx	short	mload, mmx	
<b>Note:</b> ** Bits 2, 1, and 0 of the modR/M byte select the integer register.						

Table 14. MMX Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
PMADDWD mmreg1, mmreg2	0Fh	F5h	11-xxx-xxx	short	mmx	
PMADDWD mmreg, mem64	0Fh	F5h	mm-xxx-xxx	short	mload, mmx	
PMULHW mmreg1, mmreg2	0Fh	E5h	11-xxx-xxx	short	mmx	
PMULHW mmreg, mem64	0Fh	E5h	mm-xxx-xxx	short	mload, mmx	
PMULLW mmreg1, mmreg2	0Fh	D5h	11-xxx-xxx	short	mmx	
PMULLW mmreg, mem64	0Fh	D5h	mm-xxx-xxx	short	mload, mmx	
POR mmreg1, mmreg2	0Fh	EBh	11-xxx-xxx	short	mmx	
POR mmreg, mem64	0Fh	EBh	mm-xxx-xxx	short	mload, mmx	
PSLLW mmreg1, mmreg2	0Fh	F1h	11-xxx-xxx	short	mmx	
PSLLW mmreg, mem64	0Fh	F1h	11-xxx-xxx	short	mload, mmx	
PSLLW mmreg, imm8	0Fh	71h	11-110-xxx	short	mmx	
PSLLD mmreg1, mmreg2	0Fh	F2h	11-xxx-xxx	short	mmx	
PSLLD mmreg, mem64	0Fh	F2h	11-xxx-xxx	short	mload, mmx	
PSLLD mmreg, imm8	0Fh	72h	11-110-xxx	short	mmx	
PSLLQ mmreg1, mmreg2	0Fh	F3h	11-xxx-xxx	short	mmx	
PSLLQ mmreg, mem64	0Fh	F3h	11-xxx-xxx	short	mload, mmx	
PSLLQ mmreg, imm8	0Fh	73h	11-110-xxx	short	mmx	
PSRAW mmreg1, mmreg2	0Fh	E1h	11-xxx-xxx	short	mmx	
PSRAW mmreg, mem64	0Fh	E1h	11-xxx-xxx	short	mload, mmx	
PSRAW mmreg, imm8	0Fh	71h	11-100-xxx	short	mmx	
PSRAD mmreg1, mmreg2	0Fh	E2h	11-xxx-xxx	short	mmx	
PSRAD mmreg, mem64	0Fh	E2h	11-xxx-xxx	short	mload, mmx	
PSRAD mmreg, imm8	0Fh	72h	11-100-xxx	short	mmx	
PSRAQ mmreg1, mmreg2	0Fh	E3h	11-xxx-xxx	short	mmx	
PSRAQ mmreg, mem64	0Fh	E3h	11-xxx-xxx	short	mload, mmx	
PSRAQ mmreg, imm8	0Fh	73h	11-100-xxx	short	mmx	
PSRLW mmreg1, mmreg2	0Fh	D1h	11-xxx-xxx	short	mmx	
PSRLW mmreg, mem64	0Fh	D1h	11-xxx-xxx	short	mload, mmx	
PSRLW mmreg, imm8	0Fh	71h	11-010-xxx	short	mmx	
PSRLD mmreg1, mmreg2	0Fh	D2h	11-xxx-xxx	short	mmx	
PSRLD mmreg, mem64	0Fh	D2h	11-xxx-xxx	short	mload, mmx	
<b>Note:</b> ** Bits 2, 1, and 0 of the modR/M byte select the integer register.						

# 3 Software Environment

Table 14. MMX Instructions (continued)

Instruction Mnemonic	Prefix Byte(s)	First Byte	ModR/M Byte	Decode Type	RISC86 Opcodes	Note
PSRLD mmreg, imm8	0Fh	72h	11-010-xxx	short	mmx	
PSRLQ mmreg1, mmreg2	0Fh	D3h	11-xxx-xxx	short	mmx	
PSRLQ mmreg, mem64	0Fh	D3h	11-xxx-xxx	short	mload, mmx	
PSRLQ mmreg, imm8	0Fh	73h	11-010-xxx	short	mmx	
PSUBB mmreg1, mmreg2	0Fh	F8h	11-xxx-xxx	short	mmx	
PSUBB mmreg, mem64	0Fh	F8h	mm-xxx-xxx	short	mload, mmx	
PSUBD mmreg1, mmreg2	0Fh	FAh	11-xxx-xxx	short	mmx	
PSUBD mmreg, mem64	0Fh	FAh	mm-xxx-xxx	short	mload, mmx	
PSUBSB mmreg1, mmreg2	0Fh	E8h	11-xxx-xxx	short	mmx	
PSUBSB mmreg, mem64	0Fh	E8h	mm-xxx-xxx	short	mload, mmx	
PSUBSW mmreg1, mmreg2	0Fh	E9h	11-xxx-xxx	short	mmx	
PSUBSW mmreg, mem64	0Fh	E9h	mm-xxx-xxx	short	mload, mmx	
PSUBUSB mmreg1, mmreg2	0Fh	D8h	11-xxx-xxx	short	mmx	
PSUBUSB mmreg, mem64	0Fh	D8h	mm-xxx-xxx	short	mload, mmx	
PSUBUSW mmreg1, mmreg2	0Fh	D9h	11-xxx-xxx	short	mmx	
PSUBUSW mmreg, mem64	0Fh	D9h	mm-xxx-xxx	short	mload, mmx	
PSUBW mmreg1, mmreg2	0Fh	F9h	11-xxx-xxx	short	mmx	
PSUBW mmreg, mem64	0Fh	F9h	mm-xxx-xxx	short	mload, mmx	
PUNPCKHBW mmreg1, mmreg2	0Fh	68h	11-xxx-xxx	short	mmx	
PUNPCKHBW mmreg, mem64	0Fh	68h	mm-xxx-xxx	short	mload, mmx	
PUNPCKHWD mmreg1, mmreg2	0Fh	69h	11-xxx-xxx	short	mmx	
PUNPCKHWD mmreg, mem64	0Fh	69h	mm-xxx-xxx	short	mload, mmx	
PUNPCKHDQ mmreg1, mmreg2	0Fh	6Ah	11-xxx-xxx	short	mmx	
PUNPCKHDQ mmreg, mem64	0Fh	6Ah	mm-xxx-xxx	short	mload, mmx	
PUNPCKLBW mmreg1, mmreg2	0Fh	60h	11-xxx-xxx	short	mmx	
PUNPCKLBW mmreg, mem64	0Fh	60h	mm-xxx-xxx	short	mload, mmx	
PUNPCKLWD mmreg1, mmreg2	0Fh	61h	11-xxx-xxx	short	mmx	
PUNPCKLWD mmreg, mem64	0Fh	61h	mm-xxx-xxx	short	mload, mmx	
PUNPCKLDQ mmreg1, mmreg2	0Fh	62h	11-xxx-xxx	short	mmx	
PUNPCKLDQ mmreg, mem64	0Fh	62h	mm-xxx-xxx	short	mload, mmx	
PXOR mmreg1, mmreg2	0Fh	EFh	11-xxx-xxx	short	mmx	
PXOR mmreg, mem64	0Fh	EFh	mm-xxx-xxx	short	mload, mmx	
<b>Note:</b> ** Bits 2, 1, and 0 of the modR/M byte select the integer register.						

### Table 15. 3D Instructions

79

# 3 Software Environment

**Table 15. 3D Instructions (continued)**

Instruction Mnemonic	Prefix Byte(s)	Opcode Byte	ModR/M Byte	Decode Type	RISC86® Opcodes	Note
PF2ID mmreg, mem64	0Fh, 0Fh	1Dh	mm-xxx-xxx	short	mload, 3D	
PFRCP mmreg1, mmreg2	0Fh, 0Fh	96h	11-xxx-xxx	short	3D	
PFRCP mmreg, mem64	0Fh, 0Fh	96h	mm-xxx-xxx	short	mload, 3D	
PFRSQRT mmreg1, mmreg2	0Fh, 0Fh	97h	11-xxx-xxx	short	3D	
PFRSQRT mmreg, mem64	0Fh, 0Fh	97h	mm-xxx-xxx	short	mload, 3D	
PFRCPIT1 mmreg1, mmreg2	0Fh, 0Fh	A6h	11-xxx-xxx	short	3D	
PFRCPIT1 mmreg, mem64	0Fh, 0Fh	A6h	mm-xxx-xxx	short	mload, 3D	
PFRSQIT1 mmreg1, mmreg2	0Fh, 0Fh	A7h	11-xxx-xxx	short	3D	
PFRSQIT1 mmreg, mem64	0Fh, 0Fh	A7h	mm-xxx-xxx	short	mload, 3D	
PFRCPIT2 mmreg1, mmreg2	0Fh, 0Fh	B6h	11-xxx-xxx	short	3D	
PFRCPIT2 mmreg, mem64	0Fh, 0Fh	B6h	mm-xxx-xxx	short	mload, 3D	
PMULHRW mmreg1, mmreg2	0Fh, 0Fh	B7h	11-xxx-xxx	short	mmx	3
PMULHRW mmreg1, mem64	0Fh, 0Fh	B7h	mm-xxx-xxx	short	mload, mmx	3
PREFETCH mem	0Fh	0Dh	mm-000-xxx	vector	load	1
PREFETCHW mem	0Fh	0Dh	mm-001-xxx	vector	load	1, 2

**Notes:**

- For PREFETCH and PREFETCHW, the mem8 value refers to a byte address within the 32-byte line that will be prefetched.
- PREFETCHW will be implemented in a future K86 processor. On the AMD-K6 3D processor, this instruction performs in the same manner as the PREFETCH instruction.
- The byte listed in the column titled "First Byte" is actually the immediate byte placed at the end of the instruction.

## 3D Technology

### Introduction

---

3D technology is a significant innovation to the x86 architecture that drives today's personal computers. 3D technology is a group of new instructions that opens the traditional processing bottlenecks for floating-point-intensive and multimedia applications. With 3D technology, hardware and software applications can implement more powerful solutions to create a more entertaining and productive PC platform. Examples of the type of improvements that 3D technology enables are faster frame rates on high-resolution scenes, much better physical modeling of real-world environments, sharper and more detailed 3D imaging, smoother video playback, and near theater-quality audio.

3D technology was defined and implemented in collaboration with independent software developers, including operating system designers, application developers, and graphics vendors. It is compatible with today's existing x86 software and requires no operating system support, thereby enabling 3D applications to work with all existing operating systems.

## **4** 3D Technology

---

### **Key Functionality**

---

The 3D technology instructions are intended to open a major processing bottleneck in a 3D graphics application—floating-point operations. Today's 3D applications are facing limitations due to the fact that only one floating-point execution unit exists in the most advanced x86 processors. The front end of a typical 3D graphics software pipeline performs object physics, geometry transformations, clipping, and lighting calculations. These computations are very floating-point intensive and often limit the features and functionality of a 3D application. The source of performance for the 3D instructions originates from the single instruction multiple data (SIMD) implementation. With SIMD, each instruction not only operates on two single-precision, floating-point operands, but the microarchitecture within the processor can execute up to two 3D instructions per clock through two register execution pipelines, which allows for a total of four floating-point operations per clock. In addition, because the 3D instructions use the same floating-point registers as the MMX technology instructions, task switching between MMX and 3D operations is eliminated. For more information about MMX instructions, see Appendix A, “MMX Multimedia Technology” on page 347.

The 3D technology instruction set contains 21 instructions that support SIMD floating-point operations and includes SIMD integer operations, data prefetching, and faster MMX-to-floating-point switching. To improve MPEG decoding, the 3D instructions include a specific SIMD integer instruction created to facilitate pixel-motion compensation. Because media-based software typically operates on large data sets, the processor often needs to wait for this data to be transferred from main memory. The extra time involved with retrieving this data can be avoided by using the new 3D instruction called PREFETCH. This instruction can ensure that data is in the level 1 cache when it is needed. To improve the time it takes to switch between MMX and x87 code, the 3D instructions include the FEMMS (fast entry/exit multimedia state) instruction, which eliminates much of the overhead involved with the switch. The addition of 3D technology expands the capabilities of the AMD-K6 family of processors and enables a new generation of enriched user applications.

## 3D Feature Detection

To properly identify and use the 3D instructions, the application program must determine if the processor supports them. The CPUID instruction gives programmers the ability to determine the presence of 3D technology on a processor. Software applications must first test to see if the CPUID instruction is supported. For a detailed description of the CPUID instruction, see Appendix C, “AMD Processor Recognition” on page 505.

The presence of the CPUID instruction is indicated by the ID bit (21) in the EFLAGS register. If this bit is writable, the CPUID instruction is supported. The following code sample shows how to test for the presence of the CPUID instruction.

```

pushfd                ; save EFLAGS
pop  eax              ; store EFLAGS in EAX
mov  ebx, eax         ; save in EBX for later testing
xor  eax, 00200000h   ; toggle bit 21
push eax              ; put to stack
popfd                 ; save changed EAX to EFLAGS
pushfd                ; push EFLAGS to TOS
pop  eax              ; store EFLAGS in EAX
cmp  eax, ebx         ; see if bit 21 has changed
jz   NO_CPUID         ; if no change, no CPUID

```

Once the software has identified the processor's support for CPUID, it must test for extended functions by executing extended function 8000\_000h (EAX=8000\_000h). The EAX register returns the largest extended function input value defined for the CPUID instruction on the processor. If the value is not zero, extended functions are supported.

The next step is for the programmer to determine if the 3D instructions are supported. Extended function 8000\_0001h (EAX=8000\_0001h) of the CPUID instruction provides this information. Extended function 8000\_0001h returns the feature bits in the EDX register. If bit 31 in the EDX register is set to 1, 3D instructions are supported. The following code sample shows how to test for 3D instruction support.

```

mov  eax, 8000_0001h   ; setup extended function 1
CPUID                ; call the function
test  edx, 8000_0000h  ; test 31st bit
jnz  YES_3D            ; 3D technology supported

```



## 4 3D Technology

### 3D Register Set

The complete multimedia units in the AMD-K6 3D processor combine the existing MMX instructions with the new 3D instructions. In addition, by merging 3D with MMX, it becomes possible to write x86 programs containing both integer, MMX, and floating-point graphics instructions with no performance penalty for switching between the MMX (integer) and 3D (floating-point) units.

The processor implements eight 64-bit 3D/MMX registers. These registers are mapped onto the floating-point registers. As shown in Figure 46, the 3D and MMX instructions refer to these registers as mmreg0 to mmreg7. Mapping the new 3D/MMX registers onto the floating-point register stack enables backwards compatibility for the register saving that must occur as a result of task switching.

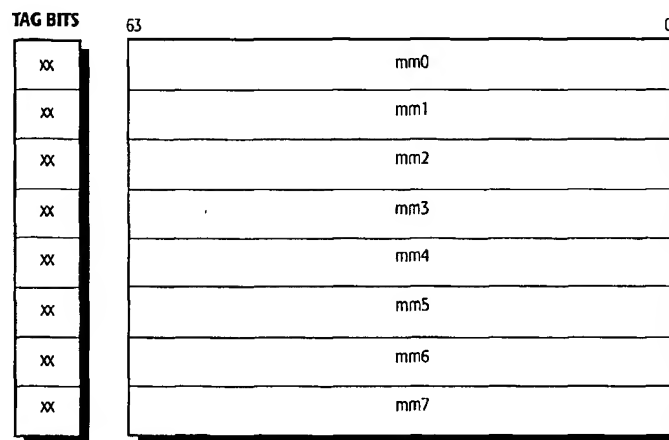


Figure 46. 3D/MMX Registers

Aliasing the 3D/MMX registers onto the floating-point register stack provides a safe method to introduce 3D and MMX technology, because it does not require modifications to existing operating systems. Instead of requiring operating system modifications, new 3D and MMX applications are supported through device drivers, 3D and MMX libraries, or

Dynamic Link Library (DLL) files. For more information, see “MMX/3D Registers” on page 31.

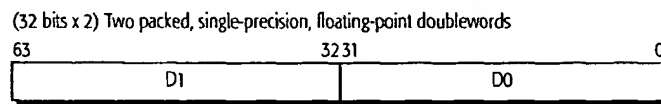
Current operating systems have support for floating-point operations and the floating-point register state. Using the floating-point registers for 3D and MMX code is a convenient way of implementing non-intrusive support for 3D and MMX instructions. Every time the processor executes an 3D or MMX instruction, all the floating-point register tag bits are set to zero (00b=valid), except for the FEMMS and EMMS instructions, which set all tag bits to one (11b=empty).

*Note: Executing the PREFETCH instruction does not change the tag bits.*

## 3D Data Type Details

3D technology uses a packed data format. The data is packed in a single, 64-bit 3D/MMX register or a quadword memory operand. For more information, see “3D Data Types” on page 32.

Figure 47 shows the 3D floating-point data type. D0 and D1 each hold an IEEE 32-bit single-precision, floating-point doubleword.

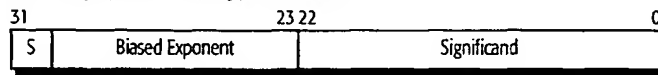


**Figure 47. 3D Data Type Details**

Figure 48 on page 86 shows the format of the IEEE 32-bit, single-precision, floating-point format.

# 4 3D Technology

32-bit, single-precision, floating-point doubleword



Value definitions

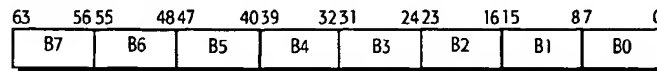
1.  $X = (-1)^S \cdot 2^{Biased\ Exponent - 127} \cdot Significand$
2.  $X = (-1)^S \cdot 2^{Biased\ Exponent - 127} \cdot Significand$  if  $0 < Biased\ Exponent < FFh$
3.  $X = Undefined$  if  $Biased\ Exponent = FFh$

X is the value of the 32-bit, single-precision, floating-point doubleword.

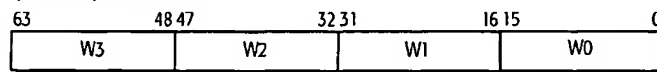
**Figure 48. Single-Precision, Floating-Point Data Format**

Figure 49 shows the formats for the integer data types.

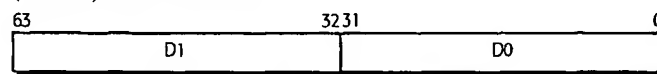
(8 bits x 8) Packed bytes



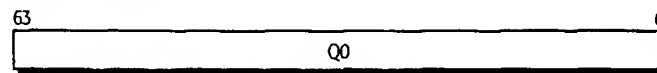
(16 bits x 4) Packed words



(32 bits x 2) Packed double words



(64 bits x 1) Quadword



**Figure 49. Integer Data Types**

## 3D Instruction Formats

The format of 3D instruction encodings is based on the conventional x86 modR/M instruction format and is similar to the format used by MMX instructions (see “MMX Instruction Formats” on page 354). The assembly language syntax used for the 3D instructions is as follows:

3D Mnemonic            mmreg1, mmreg2/mem64

The destination and source1 operand (mmreg1) must be an MMX register (mm0–mm7). The source2 operand (mmreg2/mem64) can be either an MMX register or a 64-bit memory value.

The encoding uses the opcode prefix 0Fh followed by a second opcode byte of 0Fh. To differentiate the various 3D instructions, a third instruction suffix byte is used. This suffix byte occupies the same position at the end of a 3D instructions as would an imm8 byte. The opcode format is as follows:

0Fh 0Fh modR/M [sib] [displacement] 3D\_suffix

The specific operands (mmreg1 and mmreg2/mem64) determine the values used in modR/M [sib] [displacement], and follow conventional x86 encodings. The 3D suffix is determined by the actual 3D instruction. The 3D suffixes are defined in Table 17 on page 92.

As an example, the 3D PFMUL instruction can produce the following opcodes, depending on its use:

<u>Opcode</u>	<u>Instruction</u>
0F 0F CA B4	PFMUL mm1, mm2
0F 0F 0B B4	PFMUL mm1, [ebx]
0F 0F 4B 0A B4	PFMUL mm1, [ebx+10]
26 0F 0F 0B B4	PFMUL mm1, es:[ebx]
0F 0F 4C 83 0A B4	PFMUL mm1, [ebx+eax*4+10]

The encoding of the two performance-enhancement instructions (FEMMS and PREFETCH) uses a single opcode prefix 0Fh. The details of the opcodes for these instructions are shown on pages 96 and 134 respectively.

## **4** 3D Technology

---

### **3D Definitions**

---

3D technology provides 21 additional instructions to support high-performance 3D graphics and audio processing. 3D instructions are vector instructions that operate on 64-bit registers. 3D instructions are SIMD—operating on eight 8-bit operands, four 16-bit operands, or two 32-bit operands.

The definitions for the 3D instructions starting on page 95 contain designations for the classification of the instruction as vectored or scalar. Vector instructions operate in parallel on two sets of 32-bit, single-precision, floating-point words. Instructions that are labeled as scalar instructions operate on a single set of 32-bit operands (from the low halves of the two 64-bit operands).

The 3D single-precision, floating-point format is compatible with the IEEE-754, single-precision format. This format comprises a 1-bit sign, an 8-bit biased exponent, and a 23-bit significand with one hidden integer bit for a total of 24 bits in the significand. The bias of the exponent is 127, consistent with the IEEE single-precision standard. The significands are normalized to be within the range of [1,2).

In contrast to the IEEE standard that dictates four rounding modes, 3D technology supports one rounding mode —either round-to-nearest or round-to-zero (truncation). The hardware implementation of 3D technology determines the rounding mode. The processor implements round-to-nearest mode. Regardless of the rounding mode used, the floating-point-to-integer and integer-to-floating-point conversion instructions, PF2ID and PI2FD, always use the round-to-zero (truncation) mode.

The largest-representable normal number in magnitude for this precision in hexadecimal has an exponent of FEh and a significand of 7FFFFFFh, with a numerical value of  $2^{127} (2 - 2^{-23})$ . All results that overflow above the maximum-representable positive value are saturated to either this maximum-representable normal number or to positive infinity. Similarly, all results that overflow below the minimum-representable negative value are saturated to either this minimum-representable normal number or to negative infinity.

The implementation of 3D technology determines how arithmetic overflow is handled—either properly signed maximum- or minimum-representable normal numbers or properly signed infinities. The AMD-K6 3D processor generates properly signed maximum- or minimum-representable normal numbers.

Infinities and NaNs are not supported as operands to 3D instructions.

The smallest-representable normal number in magnitude for this precision in hexadecimal has an exponent of 01h and a significand of 000000h, with a numerical value of  $2^{-126}$ . Accordingly, all results below this minimum-representable value in magnitude are held to zero. Table 16 shows the exponent ranges supported by the 3D technology.

**Table 16. 3D Technology Exponent Ranges**

Biased Exponent	Description
FFh	Unsupported *
00h	Zero
00h<x<FFh	Normal
01h	$2^{(1-127)}$ lowest possible exponent
FEh	$2^{(254-127)}$ largest possible exponent
<b>Note:</b> * Unsupported numbers can be used as operands. The results of operations with unsupported numbers are undefined.	

Like MMX instructions, 3D instructions do not generate exceptions nor do they set any status flags. It is the user's responsibility to ensure that in-range data is provided to 3D instructions and that all computations remain within valid ranges (or are held as expected).

## 3D Execution Resources

The register operations of all 3D floating-point instructions are executed by either the register X unit or the register Y unit. One operation can be issued to each register unit each clock cycle, for a maximum issue and execution rate of two 3D operations per cycle. All 3D operations have an execution latency of two clock cycles and are fully pipelined.

## 4 3D Technology

Even though 3D execution resources are not duplicated in both register units (for example, there are not two pairs of 3D multipliers, just one shared pair of multipliers), there are no instruction-decode or operation-issue pairing restrictions. When, for example, an 3D multiply operation starts execution in a register unit, that unit grabs and uses the one shared pair of 3D multipliers. Only when actual contention occurs between two 3D operations starting execution at the same time is one of the operations held up for one cycle in its first execution pipe stage while the other proceeds. The delay is never more than one cycle.

For code optimization purposes, 3D operations are grouped into two categories. These categories are based on execution resources and are important when creating properly scheduled code. As long as two 3D operations that start execution simultaneously do not fall into the same category, both operations will start execution without delay.

The first category of instructions contains the operations for the following 3D instructions: PFADD, PFSUB, PFSUBR, PFACC, PFCMPx, PFMIN, PFMAX, PI2FD, PF2ID, PFRCP, and PFRSQRT.

The second category contains the operations for the following 3D instructions: PFMUL, PFRCPIT1, PFRSQIT1, and PFRCPIT2.

*Note: 3D add and multiply operations, among other combinations, can execute simultaneously.*

Normally, in high-performance 3D code, all of the 3D instructions are properly scheduled apart from each other so as to avoid delays due to execution resource contentions (as well as taking into account dependencies and execution latencies). For further information regarding code optimization, see the Appendix B, "Code Optimization" on page 455, which provides in-depth discussions of code optimization techniques for the AMD-K6 3D processor.

The SIMD 3D instructions are summarized in Table 17 on page 92. The dedicated and shared execution resources of the register X unit and register Y unit are shown in Figure 50 on page 91. The execution resources for some MMX operations, as well as all 3D operations, are shared between the two register

units. For contention-checking purposes, each box represents a category of operations that cannot start execution simultaneously. In addition, the MMX and 3D multiplies use the same hardware, while MMX and 3D adds and subtracts do not.

The two 3D performance-enhancement instructions are summarized in Table 18 on page 92. The FEMMS instruction does not use any specific execution resource or pipeline. The PREFETCH instruction is operated on in the Load unit.

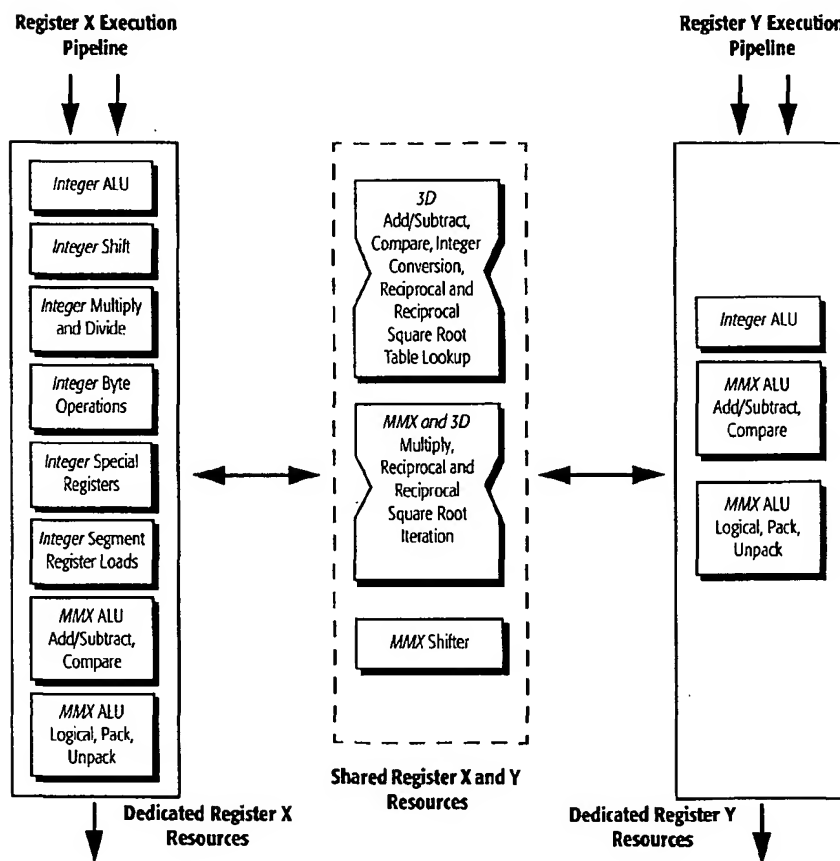


Figure 50. Register X Unit and Register Y Unit Resources



## 4 3D Technology

**Table 17. 3D Floating-Point Instructions**

Operation	Function	Opcode Suffix
PAVGUSB	Packed 8-bit Unsigned Integer Averaging	BFh
PFADD	Packed Floating-Point Addition	9Eh
PFSUB	Packed Floating-Point Subtraction	9Ah
PFSUBR	Packed Floating-Point Reverse Subtraction	AAh
PFACC	Floating-Point Accumulate	AEh
PFCMPGE	Packed Floating-Point Comparison, Greater or Equal	90h
PFCMPGT	Packed Floating-Point Comparison, Greater	A0h
PFCMPEQ	Packed Floating-Point Comparison, Equal	B0h
PFMIN	Packed Floating-Point Minimum	94h
PFMAX	Packed Floating-Point Maximum	A4h
PI2FD	Packed 32-bit Integer to Floating-Point Conversion	0Dh
PF2ID	Packed Floating-Point to 32-bit Integer	1Dh
PFRCPP	Floating-Point Reciprocal Approximation	96h
PFRSQRT	Floating-Point Reciprocal Square Root Approximation	97h
PFMUL	Packed Floating-Point Multiplication	B4h
PFRCPT1	Packed Floating-Point Reciprocal First Iteration Step	A6h
PFRSQIT1	Packed Floating-Point Reciprocal Square Root First Iteration Step	A7h
PFRCPT2	Packed Floating-Point Reciprocal/Reciprocal Square Root Second Iteration Step	B6h
PMULHRW	Packed 16-bit Integer Multiply with rounding	B7h

**Table 18. 3D Performance-Enhancement Instructions**

Operation	Function	Opcode Suffix
FEMMS	Faster entry/exit of the MMX or floating-point state	0Eh
PREFETCH	Prefetch at least a 32-byte line into L1 data cache	0Dh

Table 15 on page 79 contains a complete list of 3D instruction mnemonics.

## Task Switching

With respect to task switching, treat the 3D instructions exactly the same as MMX instructions. Operating system design must be taken into account when writing an 3D program.

The programmer must know whether the operating system automatically saves the current states when task switching, or if the 3D program has to provide the code to save states.

If a task switch occurs, the Control Register (CR0) Task Switch (TS) bit is set to 1. The processor then generates an interrupt 7 (int 7—Device Not Available) when it encounters the next floating-point, 3D, or MMX instruction, allowing the operating system to save the state of the 3D/MMX/FP registers.

In a multitasking operating system, if there is a task switch when 3D/MMX applications are running with older applications that do not include MMX instructions, the MMX/FP register state is still saved automatically through the int 7 handler. For more information, see “Task Switching” on page 356.

## 3D Exceptions

Table 19 contains a list of exceptions that 3D and MMX instructions can generate.

**Table 19. 3D and MMX Instruction Exceptions**

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)	X	X	X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

---

## 4 3D Technology

---

The rules for exceptions are the same for both MMX and 3D instructions. In addition, exception detection and handling is identical for MMX and 3D instructions. None of the exception handlers need modification.

### Notes:

1. *An invalid opcode exception (interrupt 6) occurs if an 3D instruction is executed on a processor that does not support 3D instructions.*
2. *If a floating-point exception is pending and the processor encounters an 3D instruction, FERR# is asserted and, if CR0.NE = 1, an interrupt 16 is generated. (This is the same for MMX instructions.)*

### Prefixes

The following prefixes can be used with 3D instructions:

- The segment override prefixes (2Eh/CS, 36h/SS, 3Eh/DS, 26h/ES, 64h/FS, and 65h/GS) affect 3D instructions that contain a memory operand.
- The address-size override prefix (67h) affects 3D instructions that contain a memory operand.
- The operand-size override prefix (66h) is ignored.
- The LOCK prefix (F0h) triggers an invalid opcode exception (interrupt 6).
- The REP prefixes (F3h/ REP/ REPE/ REPZ, F2h/ REPNE/ REPNZ) are ignored.

## **3D Instruction Coding**

---

For information about 3D instruction coding techniques, see “AMD-K6 3D Processor Multimedia Coding Optimizations” on page 482 in Appendix B.

## **Division and Square Root**

For information about performing division and find square roots with 3D instructions, see “Division” on page 497 and “Square Root and Reciprocal Square Root” on page 498, both in Appendix B.

## **3D Instruction Set**

---

The following 3D instruction definitions are in alphabetical order according to the instruction mnemonics.

## 4 3D Technology

### FEMMS

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
FEMMS	0F 0Eh	Faster Enter/Exit of the MMX or floating-point state

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.

Like the EMMS instruction, the FEMMS instruction can be used to clear the MMX state following the execution of a block of MMX instructions. Because the MMX registers and tag words are shared with the floating-point unit, it is necessary to clear the state before executing floating-point instructions. Unlike the EMMS instruction, the contents of the MMX/floating-point registers are undefined after a FEMMS instruction is executed. Therefore, the FEMMS instruction offers a faster context switch at the end of an MMX routine where the values in the MMX registers are no longer required. FEMMS can also be used prior to executing MMX instructions where the preceding floating-point register values are no longer required, which facilitates faster context switching.

## PAVGUSB

<i>mnemonic</i>	<i>opcode/suffix</i>	<i>description</i>
PAVGUSB mmreg1, mmreg2/mem64	0F 0Fh / BFh	Average of unsigned packed 8-bit values

Privilege: None

Registers Affected: MMX

Flags Affected: None

Exceptions Generated:

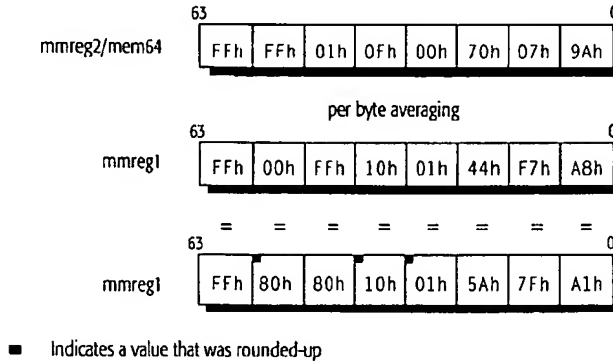
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PAVGUSB instruction produces the rounded averages of the eight unsigned 8-bit integer values in the source operand (an MMX register or a 64-bit memory location) and the eight corresponding unsigned 8-bit integer values in the destination operand (an MMX register). It does so by adding the source and destination byte values and then adding a 001h to the 9-bit intermediate value. The intermediate value is then divided by 2 (shifted right one place) and the eight unsigned 8-bit results are stored in the MMX register specified as the destination operand.

The PAVGUSB instruction can be used for pixel averaging in MPEG-2 motion compensation and video scaling operations.

# 4 3D Technology

## Functional Illustration of the PAVGUSB Instruction



The following list explains the functional illustration of the PAVGUSB instruction:

- The rounded byte average of FFh and FFh is FFh.
- The rounded byte average of FFh and 00h is 80h.
- The rounded byte average of 01h and FFh is also 80h.
- The rounded byte average of 0Fh and 10h is 10h.
- The rounded byte average of 00h and 01h is 01h.
- The rounded byte average of 70h and 44h is 5Ah.
- The rounded byte average of 07h and F7h is 7Fh.
- The rounded byte average of 9Ah and A8h is A1h.

The equations for byte averaging with rounding are as follows:

- $\text{mmreg1}[63:56] = (\text{mmreg1}[63:56] + \text{mmreg2/mem64}[63:56] + 01h)/2$
- $\text{mmreg1}[55:48] = (\text{mmreg1}[55:48] + \text{mmreg2/mem64}[55:48] + 01h)/2$
- $\text{mmreg1}[47:40] = (\text{mmreg1}[47:40] + \text{mmreg2/mem64}[47:40] + 01h)/2$
- $\text{mmreg1}[39:32] = (\text{mmreg1}[39:32] + \text{mmreg2/mem64}[39:32] + 01h)/2$
- $\text{mmreg1}[31:24] = (\text{mmreg1}[31:24] + \text{mmreg2/mem64}[31:24] + 01h)/2$
- $\text{mmreg1}[23:16] = (\text{mmreg1}[23:16] + \text{mmreg2/mem64}[23:16] + 01h)/2$
- $\text{mmreg1}[15:8] = (\text{mmreg1}[15:8] + \text{mmreg2/mem64}[15:8] + 01h)/2$
- $\text{mmreg1}[7:0] = (\text{mmreg1}[7:0] + \text{mmreg2/mem64}[7:0] + 01h)/2$

## PF2ID

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PF2ID mmreg1, mmreg2/mem64	0Fh 0Fh / 1Dh	Converts packed floating-point operand to packed 32-bit integer

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PF2ID is a vector instruction that converts a vector register containing single-precision, floating-point operands to 32-bit signed integers using truncation. Table 20 on page 100 shows the numerical range of the PF2ID instruction.

The PF2ID instruction performs the following operations:

```

IF (mmreg2/mem64[31:0] >= 231)
    THEN mmreg1[31:0] = 7FFF_FFFFh
ELSEIF (mmreg2/mem64[31:0] <= -231)
    THEN mmreg1[31:0] = 8000_0000h
ELSE mmreg1[31:0] = int(mmreg2/mem64[31:0])
IF (mmreg2/mem64[63:32] >= 231;)
    THEN mmreg1[63:32] = 7FFF_FFFFh
ELSEIF (mmreg2/mem64[63:32] <= -231)
    THEN mmreg1[63:32] = 8000_0000h
ELSE mmreg1[63:32] = int(mmreg2/mem64[63:32])

```



## 4 3D Technology

**Table 20. Numerical Range for the PF2ID Instruction**

Source 2	Source 1 and Destination
0	0
Normal, $\text{abs}(\text{Source 1}) < 1$	0
Normal, $-2147483648 < \text{Source 1} \leq -1$	round to zero (Source 1)
Normal, $1 \leq \text{Source 1} < 2147483648$	round to zero (Source 1)
Normal, $\text{Source 1} \geq 2147483648$	7FFF_FFFFh
Normal, $\text{Source 1} \leq -2147483648$	8000_0000h
Unsupported	Undefined

**Related Instructions** See the PI2FD instruction.

**PFACC**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFACC mmreg1, mmreg2/mem64	0Fh 0Fh / AEh	Floating-point accumulate

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFACC is a vector instruction that accumulates the two words of the destination operand and the source operand and stores the results in the low and high words of destination operand respectively. Both operands are single-precision, floating-point operands with 24-bit significands. Table 21 on page 102 shows the numerical range of the PFACC instruction.

The PFACC instruction performs the following operations:

$\text{mmreg1}[31:0] = \text{mmreg1}[31:0] + \text{mmreg1}[63:32]$   
 $\text{mmreg1}[63:32] = \text{mmreg2/mem64}[31:0] + \text{mmreg2/mem64}[63:32]$

**Table 21. Numerical Range for the PFACC Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/- 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the logical AND of the signs of the source operands. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 1 is used). If the absolute value of the result is greater than or equal to $2^{128}$ , the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.				

**PFADD**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFADD mmreg1, mmreg2/mem64	0Fh 0Fh / 9Eh	Packed, floating-point addition

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFADD is a vector instruction that performs addition of the destination operand and the source operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 22 on page 104 shows the numerical range of the PFADD instruction.

The PFADD instruction performs the following operations:

```

mmreg1[31:0] = mmreg1[31:0] + mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] + mmreg2/mem64[63:32]

```



Table 22. Numerical Range for the PFADD Instruction

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/- 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the logical AND of the signs of the source operands. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 1 is used). If the absolute value of the result is greater than or equal to $2^{128}$ , the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.				

**PFCMPEQ**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFCMPEQ mmreg1, mmreg2/mem64	0Fh 0Fh / B0h	Packed floating-point comparison, equal to

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFCMPEQ is a vector instruction that performs a comparison of the destination operand and the source operand and generates all one bits or all zero bits based on the result of the corresponding comparison. Table 23 on page 106 shows the numerical range of the PFCMPEQ instruction.

The PFCMPEQ instruction performs the following operations:

```

IF (mmreg1[31:0] = mmreg2/mem64[31:0])
    THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] = mmreg2/mem64[63:32])
    THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
  
```

# 4 3D Technology

**Table 23. Numerical Range for the PFCMPEQ Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	FFFF_FFFFh *	0000_0000h	0000_0000h
	Normal	0000_0000h	0000_0000h, FFFF_FFFFh **	0000_0000h
	Unsupported	0000_0000h	0000_0000h	Undefined
<b>Notes:</b> * Positive zero is equal to negative zero. ** The result is FFFF_FFFFh if source 1 and source 2 have identical signs, exponents, and mantissas. Otherwise, the result is 0000_0000h.				

**Related Instructions**      See the PCMPGE instruction.  
                                      See the PCMPGT instruction.

**PFCMPGE**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PCMPGE mmreg1, mmreg2/mem64	0Fh 0Fh / 90h	Packed floating-point comparison, greater than or equal to

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFCMPGE is a vector instruction that performs a comparison of the destination operand and the source operand and generates all one bits or all zero bits based on the result of the corresponding comparison. Table 24 on page 108 shows the numerical range of the PFCMPGE instruction.

The PFCMPGE instruction performs the following operations:

```
IF (mmreg1[31:0] >= mmreg2/mem64[31:0])
    THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] >= mmreg2/mem64[63:32])
    THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
```



**Table 24. Numerical Range for the PFCMPGE Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	FFFF_FFFFh *	0000_0000h, FFFF_FFFFh **	Undefined
	Normal	0000_0000h, FFFF_FFFFh ***	0000_0000h, FFFF_FFFFh ****	Undefined
	Unsupported	Undefined	Undefined	Undefined
<b>Notes:</b> * Positive zero is equal to negative zero. ** The result is FFFF_FFFFh, if source 2 is negative. Otherwise, the result is 0000_0000h. *** The result is FFFF_FFFFh, if source 1 is positive. Otherwise, the result is 0000_0000h. **** The result is FFFF_FFFFh, if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller than or equal in magnitude to source 2, or if source 1 and source 2 are both positive and source 1 is greater than or equal in magnitude to source 2. The result is 0000_0000h in all other cases.				

**Related Instructions**      See the PCMPEQ instruction.  
                                      See the PCMPGT instruction.

**PFCMPGT**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFCMPGT mmreg1, mmreg2/mem64	0Fh 0Fh / A0h	Packed floating-point comparison, greater than

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFCMPGT is a vector instruction that performs a comparison of the destination operand and the source operand and generates all one bits or all zero bits based on the result of the corresponding comparison. Table 25 on page 110 shows the numerical range of the PFCMPGT instruction.

The PFCMPGT instruction performs the following operations:

```

IF (mmreg1[31:0] > mmreg2/mem64[31:0])
    THEN mmreg1[31:0] = FFFF_FFFFh
ELSE mmreg1[31:0] = 0000_0000h
IF (mmreg1[63:32] > mmreg2/mem64[63:32])
    THEN mmreg1[63:32] = FFFF_FFFFh
ELSE mmreg1[63:32] = 0000_0000h
  
```

**Table 25. Numerical Range for the PFCMPGT Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	0000_0000h	0000_0000h, FFFF_FFFFh *	Undefined
	Normal	0000_0000h, FFFF_FFFF **	0000_0000h, FFFF_FFFFh ***	Undefined
	Unsupported	Undefined	Undefined	Undefined
<b>Notes:</b> * The result is FFFF_FFFFh, if source 2 is negative. Otherwise, the result is 0000_0000h. ** The result is FFFF_FFFFh, if source 1 is positive. Otherwise, the result is 0000_0000h. *** The result is FFFF_FFFFh, if source 1 is positive and source 2 is negative, or if they are both negative and source 1 is smaller in magnitude than source 2, or if source 1 and source 2 are positive and source 1 is greater in magnitude than source 2. The result is 0000_0000h in all other cases.				

**Related Instructions**      See the PCMPEQ instruction.  
                                      See the PCMPGE instruction.

**PFMAX**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFMAX mmreg1, mmreg2/mem64	0Fh 0Fh / A4h	Packed floating-point maximum

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFMAX is a vector instruction that returns the larger of the two single-precision, floating-point operands. Any operation with a zero and a negative number returns positive zero. An operation consisting of two zeros returns positive zero. Table 26 on page 112 shows the numerical range of the PFMAX instruction.

The PFMAX instruction performs the following operations:

```

IF (mmreg1[31:0] > mmreg2/mem64[31:0])
    THEN mmreg1[31:0] = mmreg1[31:0]
ELSE mmreg1[31:0] = mmreg2/mem64[31:0]
IF (mmreg1[63:32] > mmreg2/mem64[63:32])
    THEN mmreg1[63:32] = mmreg1[63:32]
ELSE mmreg1[63:32] = mmreg2/mem64[63:32]

```

**Table 26. Numerical Range for the PFMAX Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+0	Source 2, +0 *	Undefined
	Normal	Source 1, +0 **	Source 1/Source 2 ***	Undefined
	Unsupported	Undefined	Undefined	Undefined
<b>Notes:</b> * The result is source 2, if source 2 is positive. Otherwise, the result is positive zero. ** The result is source 1, if source 1 is positive. Otherwise, the result is positive zero. *** The result is source 1, if source 1 is positive and source 2 is negative. The result is source 1, if both are positive and source 1 is greater in magnitude than source 2. The result is source 1, if both are negative and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.				

**Related Instructions** See the PFMIN instruction.

## PFMIN

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFMIN mmreg1, mmreg2/mem64	0Fh 0Fh / 94h	Packed floating-point minimum

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFMIN is a vector instruction that returns the smaller of the two single-precision, floating-point operands. Any operation with a zero and a positive number returns positive zero. An operation consisting of two zeros returns positive zero. Table 27 on page 114 shows the numerical range of the PFMIN instruction.

The PFMIN instruction performs the following operations:

```

IF (mmreg1[31:0] < mmreg2/mem64[31:0])
  THEN mmreg1[31:0] = mmreg1[31:0]
ELSE mmreg1[31:0] = mmreg2/mem64[31:0]
IF (mmreg1[63:32] < mmreg2/mem64[63:32])
  THEN mmreg1[63:32] = mmreg1[63:32]
ELSE mmreg1[63:32] = mmreg2/mem64[63:32]
  
```

## 4 3D Technology

**Table 27. Numerical Range for the PFMIN Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+0	Source 2, +0 *	Undefined
	Normal	Source 1, +0 **	Source 1/Source 2 ***	Undefined
	Unsupported	Undefined	Undefined	Undefined
<b>Notes:</b> * The result is source 2, if source 2 is negative. Otherwise, the result is positive zero. ** The result is source 1, if source 1 is negative. Otherwise, the result is positive zero. *** The result is source 1, if source 1 is negative and source 2 is positive. The result is source 1, if both are negative and source 1 is greater in magnitude than source 2. The result is source 1, if both are positive and source 1 is lesser in magnitude than source 2. The result is source 2 in all other cases.				

**Related Instructions** See the PFMAX instruction.

**PFMUL**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFMUL mmreg1, mmreg2/mem64	0Fh 0Fh / B4h	Packed floating-point multiplication

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Virtual		Protected	Description
	Real	8086		
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFMUL is a vector instruction that performs multiplication of the destination operand and the source operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 28 on page 116 shows the numerical range of the PFMUL instruction.

The PFMUL instruction performs the following operations:

$\text{mmreg1}[31:0] = \text{mmreg1}[31:0] * \text{mmreg2/mem64}[31:0]$   
 $\text{mmreg1}[63:32] = \text{mmreg1}[63:32] * \text{mmreg2/mem64}[63:32]$



# 4 3D Technology

**Table 28. Numerical Range for the PFMUL Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	+/- 0 *	+/- 0 *
	Normal	+/- 0 *	Normal, +/- 0 **	Undefined
	Unsupported	+/- 0 *	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the exclusive-OR of the signs of the source operands. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the exclusive-OR of the signs of the source operands. If the absolute value of the product is greater than or equal to $2^{128}$ , the result is the largest normal number with the sign being exclusive-OR of the signs of the source operands.				

**PFRCP**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFRCP mmreg1, mmreg2/mem64	0Fh 0Fh / 96h	Floating-point reciprocal approximation

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRCP is a scalar instruction that returns a low-precision estimate of the reciprocal of the source operand. The single result value is duplicated in both high and low halves of this instruction's 64-bit result. The source operand is single-precision with a 24-bit significand, and the result is accurate to 14 bits. Table 29 on page 118 shows the numerical range of the PFRCP instruction.

Increased accuracy (the full 24 bits of a single-precision significand) requires the use of two additional instructions (PFRCPIT1 and PFRCPIT2). The first stage of this increase or refinement in accuracy (PFRCPIT1) requires that the input and output of the already executed PFRCP instruction be used as input to the PFRCPIT1 instruction. Refer to "3D Instruction Coding" on page 95 for an application-specific example of how to use this instruction and related instructions.

The PFRCP instruction performs the following operations:

```
mmreg1[31:0] = reciprocal(mmreg2/mem64[31:0])
mmreg1[63:32] = reciprocal(mmreg2/mem64[31:0])
```



## 3D Technology

In the following code example, the bold line illustrates the PFRCP instruction in a sequence used to compute  $q = a/b$  accurate to 24 bits:

```
X0 = PFRCP(b)
X1 = PFRCPIT1(b, X0)
X2 = PFRCPIT2(X1, X0)
q = PFMUL(a, X2)
```

**Table 29. Numerical Range for the PFRCP Instruction**

		Source 1 and Destination
Source 2	0	+/- Maximum Normal*
	Normal	Normal, +/- 0 **
	Unsupported	Undefined
<b>Notes:</b> * The result has the same sign as the source operand. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the sign of the source operand. Otherwise, the result is a normal with the sign being the same sign as the source operand.		

**Related Instructions**      See the PFRCPIT1 instruction.  
                                     See the PFRCPIT2 instruction.

**PFRCPIT1**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFRCPIT1 mmreg1, mmreg2/mem64	0Fh 0Fh / A6h	Packed floating-point reciprocal, first iteration step

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

**PFRCPIT1** is a vector instruction that performs the first step in a Newton-Raphson iteration to refine the reciprocal approximation produced by the PFRCP instruction (the second and final step yields a result accurate to 24 bits). Table 30 on page 120 shows the numerical range of the PFRCPIT1 instruction.

The behavior of this instruction is only defined for those combinations of operands such that one source operand was the input to the PFRCP instruction and the other source operand was the output of the same PFRCP instruction. Refer to “3D Instruction Coding” on page 95 for an application-specific example of how to use this instruction and related instructions.

In the following code example, the bold line illustrates the PFRCPIT1 instruction in a sequence used to compute  $q = a/b$  accurate to 24 bits:

```

X0 = PFRCP(b)
X1 = PFRCPIT1(b, X0)
X2 = PFRCPIT2(X1, X0)
q = PFMUL(a, X2)

```

## 4 3D Technology

Table 30. Numerical Range for the PFRCPIT1 Instruction

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	+/- 0 *	+/- 0 *
	Normal	+/- 0 *	Normal **	Undefined
	Unsupported	+/- 0 *	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the exclusive-OR of the signs of the source operands. ** The sign is positive.				

**Related Instructions**      See the PFRCP instruction.  
   See the PFRCPIT2 instruction.

## PFRCPIT2

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFRCPIT2 mmreg1, mmreg2/mem64	0Fh 0Fh / B6h	Packed floating-point reciprocal/reciprocal square root, second iteration step

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRCPIT2 is a vector instruction that performs the second and final step in a Newton-Raphson iteration to refine the reciprocal or reciprocal square root approximation produced by the PFRCP and PFSQRT instructions, respectively. Table 31 on page 122 shows the numerical range of the PFRCPIT2 instruction.

The behavior of this instruction is only defined for those combinations of operands such that the first source operand (mmreg1) was the output of either the PFRCPIT1 or PFRSQIT1 instructions and the second source operand (mmreg2/mem64) was the output of either the PFRCP or PFRSQRT instructions. Refer to “3D Instruction Coding” on page 95 for an application-specific example of how to use this instruction and related instructions.

## 4 3D Technology

In the following code example, the bold line illustrates the PFRCPIT2 instruction in a sequence used to compute  $q = a/b$  accurate to 24 bits:

```

XC = PFRCP(b)
X1 = PFRCPIT1(b, X0)
X2 = PFRCPIT2(X1, X0)
q = PFMUL(a, X2)

```

**Table 31. Numerical Range for the PFRCPIT2 Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	+/- 0 *	+/- 0 *
	Normal	+/- 0 *	Normal, +/- 0 **	Undefined
	Unsupported	+/- 0 *	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the exclusive-OR of the signs of the source operands. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the exclusive-OR of the signs of the source operands. If the absolute value of the product is greater than or equal to $2^{128}$ , the result is the largest normal number with the sign being exclusive-OR of the signs of the source operands.				

**Related Instructions**

- See the PFRCPIT1 instruction.
- See the PFRSQIT1 instruction.
- See the PFRCP instruction.
- See the PFRSQRT instruction.

**PFRSQIT1**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFRSQIT1 mmreg1, mmreg2/mem64	0Fh 0Fh / A7h	Packed floating-point reciprocal square root, first iteration step

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRSQIT1 is a vector instruction that performs the first step in a Newton-Raphson iteration to refine the reciprocal square root approximation produced by the PFSQRT instruction (the second and final step is accurate to 24 bits). Table 32 on page 124 shows the numerical range of the PFRCPIT2 instruction.

The behavior of this instruction is only defined for those combinations of operands such that one source operand was the input to the PFRSQRT instruction and the other source operand is the square of the output of the same PFRSQRT instruction. Refer to “3D Instruction Coding” on page 95 for an application-specific example of how to use this instruction and related instructions.



## 4 3D Technology

In the following code example, the bold lines illustrate the PFMUL and PFRSQIT1 instructions in a sequence used to compute  $a = 1/\sqrt{b}$  accurate to 24 bits:

```

X0 = PFRSQRT(b)
X1 = PFMUL(X0, X0)
X2 = PFRSQIT1(b, X1)
a = PFRCPIT2(X2, X0)

```

**Table 32. Numerical Range for the PFRSQIT1 Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	+/- 0 *	+/- 0 *
	Normal	+/- 0 *	Normal **	Undefined
	Unsupported	+/- 0 *	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the exclusive-OR of the signs of the source operands. ** The sign is 0.				

**Related Instructions**      See the PFRCPIT2 instruction.  
                                      See the PFRSQRT instruction.

## PFRSQRT

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFRSQRT mmreg1, mmreg2/mem64	0Fh 0Fh / 97h	Floating-point reciprocal square root approximation

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Virtual		Protected	Description
	Real	8086		
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFRSQRT is a scalar instruction that returns a low-precision estimate of the reciprocal square root of the source operand. The single result value is duplicated in both high and low halves of this instruction's 64-bit result. The source operand is single-precision with a 24-bit significand, and the result is accurate to 15 bits. Negative operands are treated as positive operands for purposes of reciprocal square root computation, with the sign of the result the same as the sign of the source operand. Table 33 on page 126 shows the numerical range of the PFRSQRT instruction.

Increased accuracy (the full 24 bits of a single-precision significand) requires the use of two additional instructions (PFRSQIT1 and PFRCPIT2). The first stage of this increase or refinement in accuracy (PFRSQIT1) requires that the input and squared output of the already executed PFRSQRT instruction be used as input to the PFRSQIT1 instruction. Refer to "3D Instruction Coding" on page 95 for an application-specific example of how to use this instruction and related instructions.

## 4 3D Technology

The PFRSQRT instruction performs the following operations:

mmreg1[31:0] = reciprocal square root(mmreg2/mem64[31:0])  
mmreg1[63:32] = reciprocal square root(mmreg2/mem64[31:0])

In the following code example, the bold line illustrates the PFRSQRT instruction in a sequence used to compute  $a = 1/\sqrt{b}$  accurate to 24 bits:

```

X0 = PFRSQRT(b)
X1 = PFMUL(X0, X0)
X2 = PFRSQIT1(b, X1)
X3 = PFRCPIT2(X2, X0)
a = PFMUL(b, X3)

```

**Table 33. Numerical Range for PFRSQRT Instruction**

		Source 1 and Destination
Source 2	0	+/- Maximum Normal*
	Normal	Normal *
	Unsupported	Undefined *
<b>Notes:</b> * The result has the same sign as the source operand.		

**Related Instructions**      See the PFRSQIT1 instruction.  
                                     See the PFRCPIT2 instruction.

**PFSUB**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFSUB mmreg1, mmreg2/mem64	0Fh 0Fh / 9Ah	Packed floating-point subtraction

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFSUB is a vector instruction that performs subtraction of the source operand from the destination operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 34 on page 128 shows the numerical range of the PFSUB instruction.

The PFSUB instruction performs the following operations:

```
mmreg1[31:0] = mmreg1[31:0] - mmreg2/mem64[31:0]
mmreg1[63:32] = mmreg1[63:32] - mmreg2/mem64[63:32]
```

# 4 3D Technology

**Table 34. Numerical Range for the PFSUB Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/- 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the logical AND of the sign of source 1 and the inverse of the sign of source 2. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 1 is used). If the absolute value of the result is greater than or equal to $2^{128}$ , the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.				

**Related Instructions** See the PFSUBR instruction.

**PFSUBR**

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PFSUBR mmreg1, mmreg2/mem64	0Fh 0Fh / AAh	Packed floating-point reverse subtraction

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PFSUBR is a vector instruction that performs subtraction of the destination operand from the source operand. Both operands are single-precision, floating-point operands with 24-bit significands. Table 35 on page 130 shows the numerical range of the PFSUBR instruction.

The PFSUBR instruction performs the following operations:

$\text{mmreg1}[31:0] = \text{mmreg2/mem64}[31:0] - \text{mmreg1}[31:0]$

$\text{mmreg1}[63:32] = \text{mmreg2/mem64}[63:32] - \text{mmreg1}[63:32]$

## 4 3D Technology

**Table 35. Numerical Range for the PFSUBR Instruction**

		Source 2		
		0	Normal	Unsupported
Source 1 and Destination	0	+/- 0 *	Source 2	Source 2
	Normal	Source 1	Normal, +/- 0 **	Undefined
	Unsupported	Source 1	Undefined	Undefined
<b>Notes:</b> * The sign of the result is the logical AND of the sign of source 1 and the inverse of the sign of source 2. ** If the absolute value of the result is less than $2^{-126}$ , the result is zero with the sign being the sign of the source operand that is larger in magnitude (if the magnitudes are equal, the sign of source 2 is used). If the absolute value of the result is greater than or equal to $2^{126}$ , the result is the largest normal number with the sign being the sign of the source operand that is larger in magnitude.				

**Related Instructions** See the PFSUB instruction.

## PI2FD

<i>mnemonic</i>	<i>opcode/imm8</i>	<i>description</i>
PI2FD mmreg1, mmreg2/mem64	0Fh 0Fh / 0Dh	Packed 32-bit integer to floating-point conversion

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

PI2FD is a vector instruction that converts a vector register containing signed, 32-bit integers to single-precision, floating-point operands. When PI2FD converts an input operand with more significant digits than are available in the output, the output is truncated.

The PI2FD instruction performs the following operations:

```
mmreg1[31:0] = float(mmreg2/mem64[31:0])
mmreg1[63:32] = float(mmreg2/mem64[63:32])
```

**Related Instructions** See the PF2ID instruction.



# 4 3D Technology

## PMULHRW

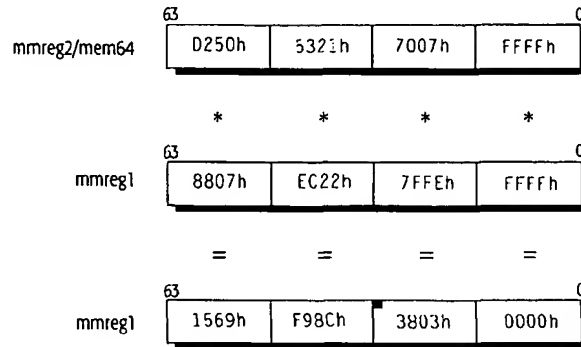
<i>mnemonic</i>	<i>opcode/suffix</i>	<i>description</i>
PMULHRW mmreg1, mmreg2/mem64	0F 0Fh/B7h	Multiply signed packed 16-bit values with rounding and store the high 16 bits.

Privilege: None  
 Registers Affected: MMX  
 Flags Affected: None  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PMULHRW instruction multiplies the four signed 16-bit integer values in the source operand (an MMX register or a 64-bit memory location) by the four corresponding signed 16-bit integer values in the destination operand (an MMX register). The PMULHRW instruction then adds 8000h to the lower 16 bits of the 32-bit result, which results in the rounding of the high-order, 16-bit result. The high-order 16 bits of the result (including the sign bit) are stored in the destination operand.

The PMULHRW instruction provides a numerically more accurate result than the PMULMH instruction, which truncates the result instead of rounding.

**Functional Illustration of the PMULHRW Instruction**

■ Indicates a value that was rounded-up

The following list explains the functional illustration of the PMULHRW instruction:

- The signed 16-bit negative value D250h (–2DB0h) is multiplied by the signed 16-bit negative value 8807h (–77F9h) to produce the signed 32-bit positive result of 1569\_4030h. 8000h is then added to the lower 16 bits to produce a final result of 1569\_C030h. This rounding does not affect the final result of 1569h. The signed high-order 16 bits of the result are stored in the destination operand.
- The signed 16-bit positive value 5321h is multiplied by the signed 16-bit negative value EC22h (–13DEh) to produce the signed 32-bit negative result of F98C\_7662h (–0673\_899Eh). 8000h is then added to the lower 16 bits, producing a final result of F98C\_F662h. This rounding does not affect the final result of F98Ch. The signed high-order 16 bits of the result are stored in the destination operand.
- The signed 16-bit positive value 7007h is multiplied by the signed 16-bit positive value 7FFEh to produce the signed 32-bit positive result of 3802\_9FF2h. 8000h is then added to the lower 16 bits to produce a final result of 3803\_1FF2h. This result has been rounded up. The signed high-order 16 bits of the result (3803h) are stored in the destination operand.
- The signed 16-bit negative value FFFFh (–1) is multiplied by the signed 16-bit negative value FFFFh (–1) to produce the signed 32-bit positive result of 0000\_0001h. 8000h is then added to the lower 16 bits to produce a final result of 0000\_8001h. This rounding does not affect the final result of 0000h. The signed high-order 16 bits of the result are stored in the destination operand.

# 4 3D Technology

## PREFETCH/PREFETCHW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PREFETCH(W) mem8	0F 0Dh	Prefetch processor cache line into L1 data cache

Privilege: none  
Registers Affected: none  
Flags Affected: none  
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.

The PREFETCH instruction loads a processor cache line into the data cache. The address of this line is specified by the mem8 value. For the AMD-K6 3D processor, the line size is 32-bytes. In all future processors, the size of the line that is loaded by the PREFETCH instruction will be at least 32-bytes. The PREFETCH instruction loads a line even if the mem8 address is not aligned with the start of the line. Although some implementations, including the AMD-K6 family of processors, may perform the cache fill starting from the cache miss or mem8 address. If a cache hit occurs (the line is already in the data cache) or a memory fault is detected, no bus cycle is initiated and the instruction is treated as a NOP.

In applications where a large number of data sets must be processed, the PREFETCH instruction can pre-load the next data set into the data cache while, simultaneously, the processor is operating on the present set of data. This instruction allows the programmer to explicitly code operation concurrency. When the present set of data values is completed, the next set is already available in the data cache. An example of a concurrent operation is vertices processing in 3D transformations, where the next set of vertices can be prefetched into the data cache while the present set is being transformed.

The PREFETCH instruction format in the processor is defined to allow extensions in future AMD K86 processors. The instruction mnemonic for the PREFETCH instruction includes the modR/M byte. Only the memory form of modR/M is valid (use of the register form results in an invalid opcode exception). Because there is no destination register, the three destination register field bits of the modR/M byte are used to define the type of prefetch to be performed. The PREFETCH and

PREFETCHW instructions are defined by the bit pattern 000b and 001b, respectively. All other bit patterns are reserved for future use.

The PREFETCHW instruction will load the prefetched line and set the cache line MESI state to modified (in anticipation of subsequent data writes to the line), unlike the PREFETCH instruction, which typically sets the state to exclusive. If the data that is prefetched into the data cache is to be modified, use of the PREFETCHW instruction will save the cycle that the PREFETCH instruction requires for modifying the data cache line state. The AMD-K6 3D processor treats a PREFETCHW instruction the same as a PREFETCH instruction. The PREFETCHW instruction should be used when the programmer expects that the data in the cache line will be modified. Otherwise, the PREFETCH instruction should be used.

Table 36 summarizes the PREFETCH type options:

**Table 36. Summary of PREFETCH Instruction Type Options**

Mod R/M	Result
11-xxx-xxx	Invalid Opcode
mm-000-xxx	PREFETCH
mm-001-xxx	PREFETCHW
mm-010-xxx	Reserved
mm-011-xxx	Reserved
mm-100-xxx	Reserved
mm-101-xxx	Reserved
mm-110-xxx	Reserved
mm-111-xxx	Reserved

**Note:** The “Reserved” PREFETCH types do not result in an Invalid Opcode Exception if executed. Instead, for forward compatibility with future processors that may implement additional forms of the PREFETCH instruction, all “Reserved” PREFETCH types are implemented as synonyms for the basic PREFETCH type (for example, the PREFETCH instruction with type 000b).

## **4** *3D Technology*

---

# 5

## Signal Descriptions

This chapter describes the signals used by the AMD-K6 3D processor. Figure 51 on page 138 shows the signals grouped by function. The arrows in the figure indicate the direction of the signal, either into or out of the processor. Signals with double-headed arrows are bidirectional. Signals with pound signs (#) are asserted Low. For more information, see “Signal Terminology” on page 139.

# 5 Signal Descriptions

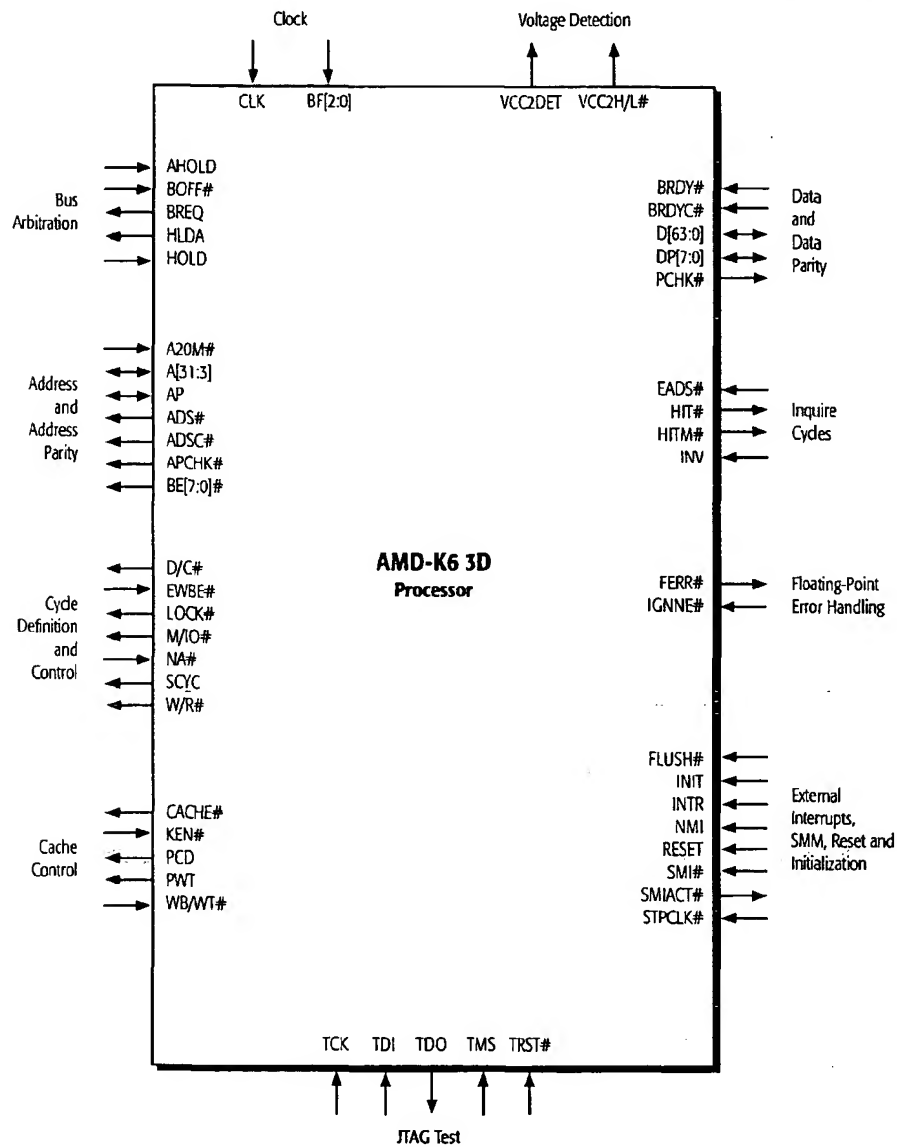


Figure 51. Logic Symbol Diagram

## Signal Terminology

---

The following terminology is used in this chapter:

- *Driven*—The processor actively pulls the signal up to the High-voltage state or pulls the signal down to the Low-voltage state.
- *Floated*—The signal is not being driven by the processor (high-impedance state), which allows another device to drive this signal.
- *Asserted*—For all active-High signals, the term *asserted* means the signal is in the High-voltage state. For all active-Low signals, the term *asserted* means the signal is in the Low-voltage state.
- *Negated*—For all active-High signals, the term *negated* means the signal is in the Low-voltage state. For all active-Low signals, the term *negated* means the signal is in the High-voltage state.
- *Sampled*—The processor has measured the state of a signal at predefined points in time and will take the appropriate action based on the state of the signal. If a signal is not sampled by the processor, its assertion or negation has no effect on the operation of the processor.



## **5** *Signal Descriptions*

### **A20M# (Address Bit 20 Mask)**

---

#### **Input**

##### **Summary**

A20M# is used to simulate the behavior of the 8086 when running in Real mode. The assertion of A20M# causes the processor to force bit 20 of the physical address to 0 prior to accessing the cache or driving out a memory bus cycle. The clearing of address bit 20 maps addresses that extend above the 8086 1-Mbyte limit to below 1 Mbyte.

##### **Sampled**

The processor samples A20M# as a level-sensitive input on every clock edge. The system logic can drive the signal either synchronously or asynchronously. If it is asserted asynchronously, it must be asserted for a minimum pulse width of two clocks.

The following list explains the effects of the processor sampling A20M# asserted under various conditions:

- Inquire cycles and writeback cycles are not affected by the state of A20M#.
- The assertion of A20M# in System Management Mode (SMM) is ignored.
- When A20M# is sampled asserted in Protected mode, it causes unpredictable processor operation. A20M# is only defined in Real mode.
- To ensure that A20M# is recognized before the first ADS# occurs following the negation of RESET, A20M# must be sampled asserted on the same clock edge that RESET is sampled negated or on one of the two subsequent clock edges.
- To ensure A20M# is recognized before the execution of an instruction, a serializing instruction must be executed between the instruction that asserts A20M# and the targeted instruction.

**A[31:3] (Address Bus)**

---

**A[31:5] Bidirectional, A[4:3] Output****Summary**

A[31:3] contain the physical address for the current bus cycle. The processor drives addresses on A[31:3] during memory and I/O cycles, and cycle definition information during special bus cycles. The processor samples addresses on A[31:5] during inquire cycles.

**Driven, Sampled, and Floated**

*As Outputs:* A[31:3] are driven valid off the same clock edge as ADS# and remain in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted. A[31:3] are driven during memory cycles, I/O cycles, special bus cycles, and interrupt acknowledge cycles. The processor continues to drive the address bus while the bus is idle.

*As Inputs:* The processor samples A[31:5] during inquire cycles on the clock edge on which EADS# is sampled asserted. Even though A4 and A3 are not used during the inquire cycle, they must be driven to a valid state and must meet the same timings as A[31:5].

A[31:3] are floated off the clock edge that AHOLD or BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.

The processor resumes driving A[31:3] off the clock edge on which the processor samples AHOLD or BOFF# negated and off the clock edge on which the processor negates HLDA.

## **5** *Signal Descriptions*

---

### **ADS# (Address Strobe)**

---

#### **Output**

##### **Summary**

The assertion of ADS# indicates the beginning of a new bus cycle. The address bus and all cycle definition signals corresponding to this bus cycle are driven valid off the same clock edge as ADS#.

##### **Driven and Floated**

ADS# is asserted for one clock at the beginning of each bus cycle. For non-pipelined cycles, ADS# can be asserted as early as the clock edge after the clock edge on which the last expected BRDY# of the cycle is sampled asserted, resulting in a single idle state between cycles. For pipelined cycles if the processor is prepared to start a new cycle, ADS# can be asserted as early as one clock edge after NA# is sampled asserted.

If AHOLD is sampled asserted, ADS# is only driven in order to perform a writeback cycle due to an inquire cycle that hits a modified cache line.

The processor floats ADS# off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.

### **ADSC# (Address Strobe Copy)**

---

#### **Output**

##### **Summary**

ADSC# has the identical function and timing as ADS#. In the event ADS# becomes too heavily loaded due to a large fanout in a system, ADSC# can be used to split the load across two outputs, which can improve system timing.

**AHOLD (Address Hold)**

---

**Input****Summary**

AHOLD can be asserted by the system to initiate one or more inquire cycles. To allow the system to drive the address bus during an inquire cycle, the processor floats A[31:3] and AP off the clock edge on which AHOLD is sampled asserted. The data bus and all other control and status signals remain under the control of the processor and are not floated. This allows a bus cycle that is in progress when AHOLD is sampled asserted to continue to completion. The processor resumes driving the address bus off the clock edge on which AHOLD is sampled negated.

If AHOLD is sampled asserted, ADS# is only asserted in order to perform a writeback cycle due to an inquire cycle that hits a modified cache line.

**Sampled**

The processor samples AHOLD on every clock edge. AHOLD is recognized while INIT and RESET are sampled asserted.

## 5 Signal Descriptions

---

### AP (Address Parity)

---

#### Bidirectional

##### *Summary*

AP contains the even parity bit for cache line addresses driven and sampled on A[31:5]. Even parity means that the total number of 1 bits on AP and A[31:5] is even. (A4 and A3 are not used for the generation or checking of address parity because these bits are not required to address a cache line.) AP is driven by the processor during processor-initiated cycles and is sampled by the processor during inquire cycles. If AP does not reflect even parity during an inquire cycle, the processor asserts APCHK# to indicate an address bus parity check. The processor does not take an internal exception as the result of detecting an address bus parity check, and system logic must respond appropriately to the assertion of this signal.

##### *Driven, Sampled, and Floated*

*As an Output:* The processor drives AP valid off the clock edge on which ADS# is asserted until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted. AP is driven during memory cycles, I/O cycles, special bus cycles, and interrupt acknowledge cycles. The processor continues to drive AP while the bus is idle.

*As an Input:* The processor samples AP during inquire cycles on the clock edge on which EADS# is sampled asserted.

The processor floats AP off the clock edge that AHOLD or BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.

The processor resumes driving AP off the clock edge on which the processor samples AHOLD or BOFF# negated and off the clock edge on which the processor negates HLDA.

## APCHK# (Address Parity Check)

---

### Output

#### Summary

If the processor detects an address parity error during an inquire cycle, APCHK# is asserted for one clock. The processor does not take an internal exception as the result of detecting an address bus parity check, and system logic must respond appropriately to the assertion of this signal.

The processor ensures that APCHK# does not glitch, enabling the signal to be used as a clocking source for system logic.

#### Driven

APCHK# is driven valid off the clock edge after the clock edge on which the processor samples EADS# asserted. It is negated off the next clock edge.

APCHK# is always driven except in the Tri-State Test mode.

## 5 Signal Descriptions

### BE[7:0]# (Byte Enables)

#### Output

##### Summary

BE[7:0]# are used by the processor to indicate the valid data bytes during a write cycle and the requested data bytes during a read cycle. The byte enables can be used to derive address bits A[2:0], which are not physically part of the processor's address bus. The processor checks and generates valid data parity for the data bytes that are valid as defined by the byte enables. The eight byte enables correspond to the eight bytes of the data bus as follows:

- |                  |                  |
|------------------|------------------|
| ■ BE7#: D[63:56] | ■ BE3#: D[31:24] |
| ■ BE6#: D[55:48] | ■ BE2#: D[23:16] |
| ■ BE5#: D[47:40] | ■ BE1#: D[15:8]  |
| ■ BE4#: D[39:32] | ■ BE0#: D[7:0]   |

The processor expects data to be driven by the system logic on all eight bytes of the data bus during a burst cache-line read cycle, independent of the byte enables that are asserted.

The byte enables are also used to distinguish between special bus cycles as defined in Table 44 on page 186.

##### Driven and Floated

BE[7:0]# are driven off the same clock edge as ADS# and remain in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted. BE[7:0]# are driven during memory cycles, I/O cycles, special bus cycles, and interrupt acknowledge cycles.

The processor floats BE[7:0]# off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD. Unlike the address bus, BE[7:0]# are not floated in response to AHOLD.

**BF[2:0] (Bus Frequency)****Inputs, Internal Pullups****Summary**

BF[2:0] determine the internal operating frequency of the processor. The frequency of the CLK input signal is multiplied internally by a ratio determined by the state of these signals as defined in Table 37. BF[2:0] have weak internal pullups and default to the 3.5 multiplier if left unconnected.

**Table 37. Processor-to-Bus Clock Ratios**

State of BF[2:0] Inputs	Processor-Clock to Bus-Clock Ratio
110b	2.0x
100b	2.5x
101b	3.0x
111b	3.5x
010b	4.0x
000b	4.5x
001b	5.0x
011b	5.5x

**Sampled**

BF[2:0] are sampled during the falling transition of RESET. They must meet a minimum setup time of 1.0 ms and a minimum hold time of two clocks relative to the negation of RESET.



## **5** *Signal Descriptions*

---

### **BOFF# (Backoff)**

---

#### **Input**

##### ***Summary***

If BOFF# is sampled asserted, the processor unconditionally aborts any cycles in progress and transitions to a bus hold state by floating the following signals: A[31:3], ADS#, ADSC#, AP, BE[7:0]#, CACHE#, D[63:0], D/C#, DP[7:0], LOCK#, M/IO#, PCD, PWT, SCYC, and W/R#. These signals remain floated until BOFF# is sampled negated. This allows an alternate bus master or the system to control the bus.

When BOFF# is sampled negated, any processor cycle that was aborted due to the assertion of BOFF# is restarted from the beginning of the cycle, regardless of the number of transfers that were completed. If BOFF# is sampled asserted on the same clock edge as BRDY# of a bus cycle of any length, then BOFF# takes precedence over the BRDY#. In this case, the cycle is aborted and restarted after BOFF# is sampled negated.

##### ***Sampled***

BOFF# is sampled on every clock edge. The processor floats its bus signals off the clock edge on which BOFF# is sampled asserted. These signals remain floated until the clock edge on which BOFF# is sampled negated.

BOFF# is recognized while INIT and RESET are sampled asserted.

**BRDY# (Burst Ready)****Input, Internal Pullup****Summary**

BRDY# is asserted to the processor by system logic to indicate either that the data bus is being driven with valid data during a read cycle or that the data bus has been latched during a write cycle. If necessary, the system logic can insert bus cycle wait states by negating BRDY# until it is ready to continue the data transfer. BRDY# is also used to indicate the completion of special bus cycles.

**Sampled**

BRDY# is sampled every clock edge within a bus cycle starting with the clock edge after the clock edge that negates ADS#. BRDY# is ignored while the bus is idle. The processor samples the following inputs on the clock edge on which BRDY# is sampled asserted: D[63:0], DP[7:0], and KEN# during read cycles, EWBE# during write cycles, and WB/WT# during read and write cycles. If NA# is sampled asserted prior to BRDY#, then KEN# and WB/WT# are sampled on the clock edge on which NA# is sampled asserted.

The number of times the processor expects to sample BRDY# asserted depends on the type of bus cycle, as follows:

- One time for a single-transfer cycle, a special bus cycle, or each of two cycles in an interrupt acknowledge sequence
- Four times for a burst cycle (once for each data transfer)

BRDY# can be held asserted for four consecutive clocks throughout the four transfers of the burst, or it can be negated to insert wait states.

## **5** *Signal Descriptions*

---

### **BRDYC# (Burst Ready Copy)**

---

#### **Input, Internal Pullup**

#### **Summary**

BRDYC# has the identical function as BRDY#. In the event BRDY# becomes too heavily loaded due to a large fanout or loading in a system, BRDYC# can be used to reduce this loading, which improves timing.

In addition, BRDYC# is sampled when RESET is negated to configure the drive strength of A[20:3], ADS#, HITM#, and W/R#. If BRDYC# is 0 during the falling transition of RESET, these particular outputs are configured using higher drive strengths than the standard strength. If BRDYC# is 1 during the falling transition of RESET, the standard strength is selected.

#### **Sampled**

BRDYC# is sampled every clock edge within a bus cycle starting with the clock edge after the clock edge that negates ADS#.

BRDYC# is also sampled during the falling transition of RESET. If RESET is driven synchronously, BRDYC# must meet the specified hold time relative to the negation of RESET. If RESET is driven asynchronously, the minimum setup and hold time for BRDYC# relative to the negation of RESET is two clocks.

**BREQ (Bus Request)**

---

**Output****Summary**

BREQ is asserted by the processor to request the bus in order to complete an internally pending bus cycle. The system logic can use BREQ to arbitrate among the bus participants. If the processor does not own the bus, BREQ is asserted until the processor gains access to the bus in order to begin the pending cycle or until the processor no longer needs to run the pending cycle. If the processor currently owns the bus, BREQ is asserted with ADS#. The processor asserts BREQ for each assertion of ADS# but does not necessarily assert ADS# for each assertion of BREQ.

**Driven**

BREQ is asserted off the same clock edge on which ADS# is asserted. BREQ can also be asserted off any clock edge, independent of the assertion of ADS#. BREQ can be negated one clock edge after it is asserted.

The processor always drives BREQ except in the Tri-State Test mode.

## 5 *Signal Descriptions*

---

### **CACHE# (Cacheable Access)**

---

#### **Output**

##### **Summary**

For reads, CACHE# is asserted to indicate the cacheability of the current bus cycle. In addition, if the processor samples KEN# asserted, which indicates the driven address is cacheable, the cycle is a 32-byte burst read cycle. For write cycles, CACHE# is asserted to indicate the current bus cycle is a modified cache-line writeback. KEN# is ignored during writebacks. If CACHE# is not asserted, or if KEN# is sampled negated during a read cycle, the cycle is not cacheable and defaults to a single-transfer cycle.

##### **Driven and Floated**

CACHE# is driven off the same clock edge as ADS# and remains in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted.

CACHE# is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.

### **CLK (Clock)**

---

#### **Input**

##### **Summary**

The CLK signal is the bus clock for the processor and is the reference for all signal timings under normal operation (except for TDI, TDO, TMS, and TRST#). BF[2:0] determine the internal frequency multiplier applied to CLK to obtain the processor's core operating frequency. See "BF[2:0] (Bus Frequency)" on page 147 for a list of the processor-to-bus clock ratios.

##### **Sampled**

The CLK signal must be stable a minimum of 1.0 ms prior to the negation of RESET to ensure the proper operation of the processor. See "CLK Switching Characteristics" on page 314 for details regarding the CLK specifications.

## D/C# (Data/Code)

---

### Output

#### *Summary*

The processor drives D/C# during a memory bus cycle to indicate whether it is addressing data or executable code. D/C# is also used to define other bus cycles, including interrupt acknowledge and special cycles. See Table 44 on page 186 for more details.

#### *Driven and Floated*

D/C# is driven off the same clock edge as ADS# and remains in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted. D/C# is driven during memory cycles, I/O cycles, special bus cycles, and interrupt acknowledge cycles.

D/C# is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.

## 5 Signal Descriptions

### D[63:0] (Data Bus)

#### Bidirectional

##### Summary

D[63:0] represent the processor's 64-bit data bus. Each of the eight bytes of data that comprise this bus is qualified as valid by its corresponding byte enable. See "BE[7:0]# (Byte Enables)" on page 146.

##### Driven, Sampled, and Floated

*As Outputs:* For single-transfer write cycles, the processor drives D[63:0] with valid data one clock edge after the clock edge on which ADS# is asserted and D[63:0] remain in the same state until the clock edge on which BRDY# is sampled asserted. If the cycle is a writeback—in which case four, 8-byte transfers occur—D[63:0] are driven one clock edge after the clock edge on which ADS# is asserted and are subsequently changed off the clock edge on which each BRDY# assertion of the burst cycle is sampled.

If the assertion of ADS# represents a pipelined write cycle that follows a read cycle, the processor does not drive D[63:0] until it is certain that contention on the data bus will not occur. In this case, D[63:0] are driven the clock edge after the last expected BRDY# of the previous cycle is sampled asserted.

*As Inputs:* During read cycles, the processor samples D[63:0] on the clock edge on which BRDY# is sampled asserted.

The processor always floats D[63:0] except when they are being driven during a write cycle as described above. In addition, D[63:0] are floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.

**DP[7:0] (Data Parity)****Bidirectional****Summary**

DP[7:0] are even parity bits for each valid byte of data—as defined by BE[7:0]#—driven and sampled on the D[63:0] data bus. Even parity means that the total number of 1 bits within each byte of data and its respective data parity bit is an even number. DP[7:0] are driven by the processor during write cycles and sampled by the processor during read cycles. If the processor detects bad parity on any valid byte of data during a read cycle, PCHK# is asserted for one clock beginning the clock edge after BRDY# is sampled asserted. The processor does not take an internal exception as the result of detecting a data parity check, and system logic must respond appropriately to the assertion of this signal.

The eight data parity bits correspond to the eight bytes of the data bus as follows:

- |                 |                 |
|-----------------|-----------------|
| ■ DP7: D[63:56] | ■ DP3: D[31:24] |
| ■ DP6: D[55:48] | ■ DP2: D[23:16] |
| ■ DP5: D[47:40] | ■ DP1: D[15:8]  |
| ■ DP4: D[39:32] | ■ DP0: D[7:0]   |

For systems that do not support data parity, DP[7:0] should be connected to V<sub>CC3</sub> through pullup resistors.

**Driven, Sampled, and Floated**

*As Outputs:* For single-transfer write cycles, the processor drives DP[7:0] with valid parity one clock edge after the clock edge on which ADS# is asserted and DP[7:0] remain in the same state until the clock edge on which BRDY# is sampled asserted. If the cycle is a writeback, DP[7:0] are driven one clock edge after the clock edge on which ADS# is asserted and are subsequently changed off the clock edge on which each BRDY# assertion of the burst cycle is sampled.

*As Inputs:* During read cycles, the processor samples DP[7:0] on the clock edge BRDY# is sampled asserted.

The processor always floats DP[7:0] except when they are being driven during a write cycle as described above. In addition, DP[7:0] are floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in recognition of HOLD.



## **5** *Signal Descriptions*

---

### **EADS# (External Address Strobe)**

---

#### **Input**

##### ***Summary***

System logic asserts EADS# during a cache inquire cycle to indicate that the address bus contains a valid address. EADS# can only be driven after the system logic has taken control of the address bus by asserting AHOLD or BOFF# or by receiving HLDA. The processor responds to the sampling of EADS# and the address bus by driving HIT#, which indicates if the inquired cache line exists in the processor's cache, and HITM#, which indicates if it is in the modified state.

##### ***Sampled***

If AHOLD or BOFF# is asserted by the system logic in order to execute a cache inquire cycle, the processor begins sampling EADS# two clock edges after AHOLD or BOFF# is sampled asserted. If the system logic asserts HOLD in order to execute a cache inquire cycle, the processor begins sampling EADS# two clock edges after the clock edge HLDA is asserted by the processor.

EADS# is ignored during the following conditions:

- One clock edge after the clock edge on which EADS# is sampled asserted
- Two clock edges after the clock edge on which ADS# is asserted
- When the processor is driving the address bus
- When the processor asserts HITM#

**EWBE# (External Write Buffer Empty)**

---

**Input****Summary**

The system logic can negate EWBE# to the processor to indicate that its external write buffers are full and that additional data cannot be stored at this time. This causes the processor to delay the following activities until EWBE# is sampled asserted:

- The commitment of write hit cycles to cache lines in the modified state or exclusive state in the processor's cache
- The decode and execution of an instruction that follows a currently-executing serializing instruction
- The assertion or negation of SMIACK#
- The entering of the Halt state and the Stop Grant state

Negating EWBE# does not prevent the completion of any type of cycle that is currently in progress.

**Sampled**

The processor samples EWBE# on each clock edge that BRDY# is sampled asserted during all memory write cycles (except writeback cycles), I/O write cycles, and special bus cycles.

If EWBE# is sampled negated, it is sampled on every clock edge until it is asserted, and then it is ignored until BRDY# is sampled asserted in the next write cycle or special cycle.

## 5 *Signal Descriptions*

---

### **FERR# (Floating-Point Error)**

---

#### **Output**

#### **Summary**

The assertion of FERR# indicates the occurrence of an unmasked floating-point exception resulting from the execution of a floating-point instruction. This signal is provided to allow the system logic to handle this exception in a manner consistent with IBM-compatible PC/AT systems. See "Handling Floating-Point Exceptions" on page 254 for a system logic implementation that supports floating-point exceptions.

The state of the numeric error (NE) bit in CR0 does not affect the FERR# signal.

The processor ensures that FERR# does not glitch, enabling the signal to be used as a clocking source for system logic.

#### **Driven**

The processor asserts FERR# on the instruction boundary of the next floating-point instruction, MMX instruction, or WAIT instruction that occurs following the floating-point instruction that caused the unmasked floating-point exception—that is, FERR# is not asserted at the time the exception occurs. The IGNNE# signal does not affect the assertion of FERR#.

FERR# is negated during the following conditions:

- Following the successful execution of the floating-point instructions FCLEX, FINIT, FSAVE, and FSTENV
- Under certain circumstances, following the successful execution of the floating-point instructions FLDCW, FLDENV, and FRSTOR, which load the floating-point status word or the floating-point control word
- Following the falling transition of RESET

FERR# is always driven except in the Tri-State Test mode.

See "IGNNE# (Ignore Numeric Exception)" on page 163 for more details on floating-point exceptions.

## FLUSH# (Cache Flush)

---

### Input

#### Summary

In response to sampling FLUSH# asserted, the processor writes back any data cache lines that are in the modified state, invalidates all lines in the instruction and data caches, and then executes a flush acknowledge special cycle. See Table 44 on page 186 for the bus definition of special cycles.

In addition, FLUSH# is sampled when RESET is negated to determine if the processor enters the Tri-State Test mode. If FLUSH# is 0 during the falling transition of RESET, the processor enters the Tri-State Test mode instead of performing the normal RESET functions.

#### Sampled

FLUSH# is sampled and latched as a falling edge-sensitive signal. During normal operation (not RESET), FLUSH# is sampled on every clock edge but is not recognized until the next instruction boundary. If FLUSH# is asserted synchronously, it can be asserted for a minimum of one clock. If FLUSH# is asserted asynchronously, it must have been negated for a minimum of two clocks, followed by an assertion of a minimum of two clocks.

FLUSH# is also sampled during the falling transition of RESET. If RESET and FLUSH# are driven synchronously, FLUSH# is sampled on the clock edge prior to the clock edge on which RESET is sampled negated. If RESET is driven asynchronously, the minimum setup and hold time for FLUSH#, relative to the negation of RESET, is two clocks.

## **5** *Signal Descriptions*

---

### **HIT# (Inquire Cycle Hit)**

---

#### **Output**

#### **Summary**

The processor asserts HIT# during an inquire cycle to indicate that the cache line is valid within the processor's instruction or data cache (also known as a cache hit). The cache line can be in the modified, exclusive, or shared state.

#### **Driven**

HIT# is always driven—except in the Tri-State Test mode—and only changes state the clock edge after the clock edge on which EADS# is sampled asserted. It is driven in the same state until the next inquire cycle.

### **HITM# (Inquire Cycle Hit To Modified Line)**

---

#### **Output**

#### **Summary**

The processor asserts HITM# during an inquire cycle to indicate that the cache line exists in the processor's data cache in the modified state. The processor performs a writeback cycle as a result of this cache hit. If an inquire cycle hits a cache line that is currently being written back, the processor asserts HITM# but does not execute another writeback cycle. The system logic must not expect the processor to assert ADS# each time HITM# is asserted.

#### **Driven**

HITM# is always driven—except in the Tri-State Test mode—and, in particular, is driven to represent the result of an inquire cycle the clock edge after the clock edge on which EADS# is sampled asserted. If HITM# is negated in response to the inquire address, it remains negated until the next inquire cycle. If HITM# is asserted in response to the inquire address, it remains asserted throughout the writeback cycle and is negated one clock edge after the last BRDY# of the writeback is sampled asserted.

**HLDA (Hold Acknowledge)**

---

**Output****Summary**

When HOLD is sampled asserted, the processor completes the current bus cycles, floats the processor bus, and asserts HLDA in an acknowledgment that these events have been completed. The processor does not assert HLDA until the completion of a locked sequence of cycles. While HLDA is asserted, another bus master can drive cycles on the bus, including inquire cycles to the processor. The following signals are floated when HLDA is asserted: A[31:3], ADS#, ADSC#, AP, BE[7:0]#, CACHE#, D[63:0], D/C#, DP[7:0], LOCK#, M/IO#, PCD, PWT, SCYC, and W/R#.

The processor ensures that HLDA does not glitch.

**Driven**

HLDA is always driven except in the Tri-State Test mode. If a processor cycle is in progress while HOLD is sampled asserted, HLDA is asserted one clock edge after the last BRDY# of the cycle is sampled asserted. If the bus is idle, HLDA is asserted one clock edge after HOLD is sampled asserted. HLDA is negated one clock edge after the clock edge on which HOLD is sampled negated.

The assertion of HLDA is independent of the sampled state of BOFF#.

The processor floats the bus every clock in which HLDA is asserted.

## **5** *Signal Descriptions*

---

### **HOLD (Bus Hold Request)**

---

#### **Input**

#### ***Summary***

The system logic can assert HOLD to gain control of the processor's bus. When HOLD is sampled asserted, the processor completes the current bus cycles, floats the processor bus, and asserts HLDA in an acknowledgment that these events have been completed.

#### ***Sampled***

The processor samples HOLD on every clock edge. If a processor cycle is in progress while HOLD is sampled asserted, HLDA is asserted one clock edge after the last BRDY# of the cycle is sampled asserted. If the bus is idle, HLDA is asserted one clock edge after HOLD is sampled asserted. HOLD is recognized while INIT and RESET are sampled asserted.

## IGNNE# (Ignore Numeric Exception)

---

### Input

#### Summary

IGNNE#, in conjunction with the numeric error (NE) bit in CR0, is used by the system logic to control the effect of an unmasked floating-point exception on a previous floating-point instruction during the execution of a floating-point instruction, MMX instruction, or the WAIT instruction—hereafter referred to as the target instruction.

If an unmasked floating-point exception is pending and the target instruction is considered error-sensitive, then the relationship between NE and IGNNE# is as follows:

- If NE = 0, then:
  - If IGNNE# is sampled asserted, the processor ignores the floating-point exception and continues with the execution of the target instruction.
  - If IGNNE# is sampled negated, the processor waits until it samples IGNNE#, INTR, SMI#, NMI, or INIT asserted.
    - If IGNNE# is sampled asserted while waiting, the processor ignores the floating-point exception and continues with the execution of the target instruction.
    - If INTR, SMI#, NMI, or INIT is sampled asserted while waiting, the processor handles its assertion appropriately.
- If NE = 1, the processor invokes the INT 10h exception handler.

If an unmasked floating-point exception is pending and the target instruction is considered error-insensitive, then the processor ignores the floating-point exception and continues with the execution of the target instruction.

FERR# is not affected by the state of the NE bit or IGNNE#. FERR# is always asserted at the instruction boundary of the target instruction that follows the floating-point instruction that caused the unmasked floating-point exception.

This signal is provided to allow the system logic to handle exceptions in a manner consistent with IBM-compatible PC/AT systems.



## **5** *Signal Descriptions*

---

### ***Sampled***

The processor samples IGNNE# as a level-sensitive input on every clock edge. The system logic can drive the signal either synchronously or asynchronously. If it is asserted asynchronously, it must be asserted for a minimum pulse width of two clocks.

## **INIT (Initialization)**

---

### **Input**

### ***Summary***

The assertion of INIT causes the processor to empty its pipelines, to initialize most of its internal state, and to branch to address FFFF\_FFF0h—the same instruction execution starting point used after RESET. Unlike RESET, the processor preserves the contents of its caches, the floating-point state, the MMX state, model-specific registers, the CD and NW bits of the CR0 register, and other specific internal resources.

INIT can be used as an accelerator for 80286 code that requires a reset to exit from Protected mode back to Real mode.

### ***Sampled***

INIT is sampled and latched as a rising edge-sensitive signal. INIT is sampled on every clock edge but is not recognized until the next instruction boundary. During an I/O write cycle, it must be sampled asserted a minimum of three clock edges before BRDY# is sampled asserted if it is to be recognized on the boundary between the I/O write instruction and the following instruction.

If INIT is asserted synchronously, it can be asserted for a minimum of one clock. If it is asserted asynchronously, it must have been negated for a minimum of two clocks, followed by an assertion of a minimum of two clocks.

## INTR (Maskable Interrupt)

---

### Input

#### Summary

INTR is the system's maskable interrupt input to the processor. When the processor samples and recognizes INTR asserted, the processor executes a pair of interrupt acknowledge bus cycles and then jumps to the interrupt service routine specified by the interrupt number that was returned during the interrupt acknowledge sequence. The processor only recognizes INTR if the interrupt flag (IF) in the EFLAGS register equals 1.

#### Sampled

The processor samples INTR as a level-sensitive input on every clock edge, but the interrupt request is not recognized until the next instruction boundary. The system logic can drive INTR either synchronously or asynchronously. If it is asserted asynchronously, it must be asserted for a minimum pulse width of two clocks. In order to be recognized, INTR must remain asserted until an interrupt acknowledge sequence is complete.

## INV (Invalidation Request)

---

### Input

#### Summary

During an inquire cycle, the state of INV determines whether an addressed cache line that is found in the processor's instruction or data cache transitions to the invalid state or the shared state.

If INV is sampled asserted during an inquire cycle, the processor transitions the cache line (if found) to the invalid state, regardless of its previous state. If INV is sampled negated during an inquire cycle, the processor transitions the cache line (if found) to the shared state. In either case, if the cache line is found in the modified state, the processor writes it back to memory before changing its state.

#### Sampled

INV is sampled on the clock edge on which EADS# is sampled asserted.

## **5** *Signal Descriptions*

---

### **KEN# (Cache Enable)**

---

#### **Input**

#### **Summary**

If KEN# is sampled asserted, it indicates that the address presented by the processor is cacheable. If KEN# is sampled asserted and the processor intends to perform a cache-line fill (signified by the assertion of CACHE#), the processor executes a 32-byte burst read cycle and expects to sample BRDY# asserted a total of four times. If KEN# is sampled negated during a read cycle, a single-transfer cycle is executed and the processor does not cache the data. For write cycles, CACHE# is asserted to indicate the current bus cycle is a modified cache-line writeback. KEN# is ignored during writebacks.

If PCD is asserted during a bus cycle, the processor does not cache any data read during that cycle, regardless of the state of KEN#. See "PCD (Page Cache Disable)" on page 171 for more details.

If the processor has sampled the state of KEN# during a cycle, and that cycle is aborted due to the sampling of BOFF# asserted, the system logic must ensure that KEN# is sampled in the same state when the processor restarts the aborted cycle.

#### **Sampled**

KEN# is sampled on the clock edge on which the first BRDY# or NA# of a read cycle is sampled asserted. If the read cycle is a burst, KEN# is ignored during the last three assertions of BRDY#. KEN# is sampled during read cycles only when CACHE# is asserted.

## LOCK# (Bus Lock)

---

### Output

#### Summary

The processor asserts LOCK# during a sequence of bus cycles to ensure that the cycles are completed without allowing other bus masters to intervene. Locked operations consist of two to five bus cycles. LOCK# is asserted during the following operations:

- An interrupt acknowledge sequence
- Descriptor Table accesses
- Page Directory and Page Table accesses
- XCHG instruction
- An instruction with an allowable LOCK prefix

In order to ensure that locked operations appear on the bus and are visible to the entire system, any data operands addressed during a locked cycle that reside in the processor's cache are flushed and invalidated from the cache prior to the locked operation. If the cache line is in the modified state, it is written back and invalidated prior to the locked operation. Likewise, any data read during a locked operation is not cached.

The processor ensures that LOCK# does not glitch.

#### Driven and Floated

During a locked cycle, LOCK# is asserted off the same clock edge on which ADS# is asserted and remains asserted until the last BRDY# of the last bus cycle is sampled asserted. The processor negates LOCK# for at least one clock between consecutive sequences of locked operations to allow the system logic to arbitrate for the bus.

LOCK# is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in response to HOLD. When LOCK# is floated due to BOFF# sampled asserted, the system logic is responsible for preserving the lock condition while LOCK# is in the high-impedance state.

## **5** *Signal Descriptions*

---

### **M/IO# (Memory or I/O)**

---

#### **Output**

##### ***Summary***

The processor drives M/IO# during a bus cycle to indicate whether it is addressing the memory or I/O space. If M/IO# = 1, the processor is addressing memory or a memory-mapped I/O port as the result of an instruction fetch or an instruction that loads or stores data. If M/IO# = 0, the processor is addressing an I/O port during the execution of an I/O instruction. In addition, M/IO# is used to define other bus cycles, including interrupt acknowledge and special cycles. See Table 44 on page 186 for more details.

##### ***Driven and Floated***

M/IO# is driven off the same clock edge as ADS# and remains in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted. M/IO# is driven during memory cycles, I/O cycles, special bus cycles, and interrupt acknowledge cycles.

M/IO# is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in response to HOLD.

**NA# (Next Address)**

---

**Input****Summary**

System logic asserts NA# to indicate to the processor that it is ready to accept another bus cycle pipelined into the previous bus cycle. ADS#, along with address and status signals, can be asserted as early as one clock edge after NA# is sampled asserted if the processor is prepared to start a new cycle. Because the processor allows a maximum of two cycles to be in progress at a time, the assertion of NA# is sampled while two cycles are in progress but ADS# is not asserted until the completion of the first cycle.

**Sampled**

NA# is sampled every clock edge during bus cycles, starting one clock edge after the clock edge that negates ADS#, until the last expected BRDY# of the last executed cycle is sampled asserted (with the exception of the clock edge after the clock edge that negates the ADS# for a second pending cycle). Because the processor latches NA# when sampled, the system logic only needs to assert NA# for one clock.

## **5** *Signal Descriptions*

---

### **NMI (Non-Maskable Interrupt)**

---

#### **Input**

#### **Summary**

When NMI is sampled asserted, the processor jumps to the interrupt service routine defined by interrupt number 02h. Unlike the INTR signal, software cannot mask the effect of NMI if it is sampled asserted by the processor. However, NMI is temporarily masked upon entering System Management Mode (SMM). In addition, an interrupt acknowledge cycle is not executed because the interrupt number is predefined.

If NMI is sampled asserted while the processor is executing the interrupt service routine for a previous NMI, the subsequent NMI remains pending until the completion of the execution of the IRET instruction at the end of the interrupt service routine.

#### **Sampled**

NMI is sampled and latched as a rising edge-sensitive signal. During normal operation, NMI is sampled on every clock edge but is not recognized until the next instruction boundary. If it is asserted synchronously, it can be asserted for a minimum of one clock. If it is asserted asynchronously, it must have been negated for a minimum of two clocks, followed by an assertion of a minimum of two clocks.

**PCD (Page Cache Disable)****Output****Summary**

The processor drives PCD to indicate the operating system's specification of cacheability for the page being addressed. System logic can use PCD to control external caching. If PCD is asserted, the addressed page is not cached. If PCD is negated, the cacheability of the addressed page depends upon the state of CACHE# and KEN#.

The state of PCD depends upon the processor's operating mode and the state of certain bits in its control registers and TLB as follows:

- In Real mode, or in Protected and Virtual-8086 modes while paging is disabled (PG bit in CR0 set to 0):  
PCD output = CD bit in CR0
- In Protected and Virtual-8086 modes while caching is enabled (CD bit in CR0 set to 0) and paging is enabled (PG bit in CR0 set to 1):
  - For accesses to I/O space, page directory entries, and other non-paged accesses:  
PCD output = PCD bit in CR3
  - For accesses to 4-Kbyte page table entries or 4-Mbyte pages:  
PCD output = PCD bit in page directory entry
  - For accesses to 4-Kbyte pages:  
PCD output = PCD bit in page table entry

**Driven and Floated**

PCD is driven off the same clock edge as ADS# and remains in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted.

PCD is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in response to HOLD.



---

## **5** *Signal Descriptions*

---

### **PCHK# (Parity Check)**

---

#### **Output**

##### **Summary**

The processor asserts PCHK# during read cycles if it detects an even parity error on one or more valid bytes of D[63:0] during a read cycle. (Even parity means that the total number of 1 bits within each byte of data and its respective data parity bit is even.) The processor checks data parity for the data bytes that are valid, as defined by BE[7:0]#, the byte enables.

PCHK# is always driven but is only asserted for memory and I/O read bus cycles and the second cycle of an interrupt acknowledge sequence. PCHK# is not driven during any type of write cycles or special bus cycles. The processor does not take an internal exception as the result of detecting a data parity error, and system logic must respond appropriately to the assertion of this signal.

The processor ensures that PCHK# does not glitch, enabling the signal to be used as a clocking source for system logic.

##### **Driven**

PCHK# is always driven except in the Tri-State Test mode. For each BRDY# returned to the processor during a read cycle with a parity error detected on the data bus, PCHK# is asserted for one clock, one clock edge after BRDY# is sampled asserted.

## PWT (Page Writethrough)

### Output

#### Summary

The processor drives PWT to indicate the operating system's specification of the writeback state or writethrough state for the page being addressed. PWT, together with WB/WT#, specifies the data cache-line state during cacheable read misses and write hits to shared cache lines. See "WB/WT# (Writeback or Writethrough)" on page 183 for more details.

The state of PWT depends upon the processor's operating mode and the state of certain bits in its control registers and TLB as follows:

- In Real mode, or in Protected and Virtual-8086 modes while paging is disabled (PG bit in CR0 set to 0):  
PWT output = 0 (writeback state)
- In Protected and Virtual-8086 modes while paging is enabled (PG bit in CR0 set to 1):
  - For accesses to I/O space, page directory entries, and other non-paged accesses:  
PWT output = PWT bit in CR3
  - For accesses to 4-Kbyte page table entries or 4-Mbyte pages:  
PWT output = PWT bit in page directory entry
  - For accesses to 4-Kbyte pages:  
PWT output = PWT bit in page table entry

#### Driven and Floated

PWT is driven off the same clock edge as ADS# and remains in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted.

PWT is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in response to HOLD.

## **5** *Signal Descriptions*

---

### **RESET (Reset)**

---

#### **Input**

##### **Summary**

When the processor samples RESET asserted, it immediately flushes and initializes all internal resources and its internal state including its pipelines and caches, the floating-point state, the MMX state, the 3D state, and all registers, and then the processor jumps to address FFFF\_FFF0h to start instruction execution.

The signals BRDYC# and FLUSH# are sampled during the falling transition of RESET to select the drive strength of selected output signals and to invoke the Tri-State Test mode, respectively. See these signal descriptions for more details.

##### **Sampled**

RESET is sampled as a level-sensitive input on every clock edge. System logic can drive the signal either synchronously or asynchronously.

During the initial power-on reset of the processor, RESET must remain asserted for a minimum of 1.0 ms after CLK and V<sub>CC</sub> reach specification before it is negated.

During a warm reset, while CLK and V<sub>CC</sub> are within their specification, RESET must remain asserted for a minimum of 15 clocks prior to its negation.

## RSVD (Reserved)

---

### Summary

Reserved signals are a special class of pins that can be treated in one of the following ways:

- As no-connect (NC) pins, in which case these pins are left unconnected
- As pins connected to the system logic as defined by the industry-standard Pentium interface (Socket 7)
- Any combination of NC and Socket 7 pins

In any case, if the RSVD pins are treated accordingly, the normal operation of the AMD-K6 3D processor is not adversely affected in any manner.

See "Pin Designations" on page 342 for a list of the locations of the RSVD pins.

## SCYC (Split Cycle)

---

### Output

### Summary

The processor asserts SCYC during misaligned, locked transfers on the D[63:0] data bus. The processor generates additional bus cycles to complete the transfer of misaligned data.

For purposes of bus cycles, the term *aligned* means:

- Any 1-byte transfers
- 2-byte and 4-byte transfers that lie within 4-byte address boundaries
- 8-byte transfers that lie within 8-byte address boundaries

### Driven and Floated

SCYC is asserted off the same clock edge as ADS#, and negated off the clock edge on which NA# or the last expected BRDY# of the entire locked sequence is sampled asserted. SCYC is only valid during locked memory cycles.

SCYC is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in response to HOLD.

## **5** *Signal Descriptions*

---

### **SMI# (System Management Interrupt)**

---

#### **Input, Internal Pullup**

##### **Summary**

The assertion of SMI# causes the processor to enter System Management Mode (SMM). Upon recognizing SMI#, the processor performs the following actions, in the order shown:

1. Flushes its instruction pipelines
2. Completes all pending and in-progress bus cycles
3. Acknowledges the interrupt by asserting SMIACK# after sampling EWBE# asserted
4. Saves the internal processor state in SMM memory
5. Disables interrupts by clearing the interrupt flag (IF) in EFLAGS and disables NMI interrupts
6. Jumps to the entry point of the SMM service routine at the SMM base physical address which defaults to 0003\_8000h in SMM memory

See Chapter 3, "System Management Mode (SMM)" on page 257 for more details regarding SMM.

##### **Sampled**

SMI# is sampled and latched as a falling edge-sensitive signal. SMI# is sampled on every clock edge but is not recognized until the next instruction boundary. If SMI# is to be recognized on the instruction boundary associated with a BRDY#, it must be sampled asserted a minimum of three clock edges before the BRDY# is sampled asserted. If it is asserted synchronously, it can be asserted for a minimum of one clock. If it is asserted asynchronously, it must have been negated for a minimum of two clocks followed by an assertion of a minimum of two clocks.

A second assertion of SMI# while in SMM is latched but is not recognized until the SMM service routine is exited.

**SMIACT# (System Management Interrupt Active)**

---

**Output****Summary**

The processor acknowledges the assertion of SMI# with the assertion of SMIACT# to indicate that the processor has entered System Management Mode (SMM). The system logic can use SMIACT# to enable SMM memory. See "SMI# (System Management Interrupt)" on page 176 for more details.

See Chapter 3, "System Management Mode (SMM)" on page 257 for more details regarding SMM.

**Driven**

The processor asserts SMIACT# after the last BRDY# of the last pending bus cycle is sampled asserted (including all pending write cycles) and after EWBE# is sampled asserted. SMIACT# remains asserted until after the last BRDY# of the last pending bus cycle associated with exiting SMM is sampled asserted.

SMIACT# remains asserted during any flush, internal snoop, or writeback cycle due to an inquire cycle.

## **5** *Signal Descriptions*

---

### **STPCLK# (Stop Clock)**

---

#### **Input, Internal Pullup**

##### **Summary**

The assertion of STPCLK# causes the processor to enter the Stop Grant state, during which the processor's internal clock is stopped. From the Stop Grant state, the processor can subsequently transition to the Stop Clock state, in which the bus clock CLK is stopped. Upon recognizing STPCLK#, the processor performs the following actions, in the order shown:

1. Flushes its instruction pipelines
2. Completes all pending and in-progress bus cycles
3. Acknowledges the STPCLK# assertion by executing a Stop Grant special bus cycle (see Table 44 on page 186)
4. Stops its internal clock after BRDY# of the Stop Grant special bus cycle is sampled asserted and after EWBE# is sampled asserted
5. Enters the Stop Clock state if the system logic stops the bus clock CLK (optional)

See Chapter 12, "Clock Control" on page 291 for more details regarding clock control.

##### **Sampled**

STPCLK# is sampled as a level-sensitive input on every clock edge but is not recognized until the next instruction boundary. System logic can drive the signal either synchronously or asynchronously. If it is asserted asynchronously, it must be asserted for a minimum pulse width of two clocks.

STPCLK# must remain asserted until recognized, which is indicated by the completion of the Stop Grant special cycle.

## TCK (Test Clock)

---

### Input, Internal Pullup

**Summary**

TCK is the clock for boundary-scan testing using the Test Access Port (TAP). See “Boundary-Scan Test Access Port (TAP)” on page 271 for details regarding the operation of the TAP controller.

**Sampled**

The processor always samples TCK, except while TRST# is asserted.

## TDI (Test Data Input)

---

### Input, Internal Pullup

**Summary**

TDI is the serial test data and instruction input for boundary-scan testing using the Test Access Port (TAP). See “Boundary-Scan Test Access Port (TAP)” on page 271 for details regarding the operation of the TAP controller.

**Sampled**

The processor samples TDI on every rising TCK edge but only while in the Shift-IR and Shift-DR states.

## TDO (Test Data Output)

---

### Output

**Summary**

TDO is the serial test data and instruction output for boundary-scan testing using the Test Access Port (TAP). See “Boundary-Scan Test Access Port (TAP)” on page 271 for details regarding the operation of the TAP controller.

**Driven and Floated**

The processor drives TDO on every falling TCK edge but only while in the Shift-IR and Shift-DR states. TDO is floated at all other times.



## **5** *Signal Descriptions*

---

### **TMS (Test Mode Select)**

---

#### **Input, Internal Pullup**

<i>Summary</i>	TMS specifies the test function and sequence of state changes for boundary-scan testing using the Test Access Port (TAP). See “Boundary-Scan Test Access Port (TAP)” on page 271 for details regarding the operation of the TAP controller.
<i>Sampled</i>	The processor samples TMS on every rising TCK edge. If TMS is sampled High for five or more consecutive clocks, the TAP controller enters its Test-Logic-Reset state, regardless of the controller state. This action is the same as that achieved by asserting TRST#.

### **TRST# (Test Reset)**

---

#### **Input, Internal Pullup**

<i>Summary</i>	The assertion of TRST# initializes the Test Access Port (TAP) by resetting its state machine to the Test-Logic-Reset state. See “Boundary-Scan Test Access Port (TAP)” on page 271 for details regarding the operation of the TAP controller.
<i>Sampled</i>	TRST# is a completely asynchronous input that does not require a minimum setup and hold time relative to TCK. See Table 80 on page 326 for the minimum pulse width requirement.

**VCC2DET (V<sub>CC2</sub> Detect)****Output****Summary**

VCC2DET is internally tied to V<sub>SS</sub> (logic level 0) to indicate to the system logic that it must supply the specified dual-voltage requirements to the V<sub>CC2</sub> and V<sub>CC3</sub> pins. The V<sub>CC2</sub> pins supply voltage to the processor core, independent of the voltage supplied to the I/O buffers on the V<sub>CC3</sub> pins. Upon sampling VCC2DET Low, system logic should sample VCC2H/L# to identify core voltage requirements.

**Driven**

VCC2DET always equals 0 and is never floated—even during the Tri-State Test mode.

**VCC2H/L# (V<sub>CC2</sub> High/Low)****Output****Summary**

VCC2H/L# is internally tied to V<sub>SS</sub> (logic level 0) to indicate to the system logic that it must supply the specified processor core voltage to the V<sub>CC2</sub> pins. The V<sub>CC2</sub> pins supply voltage to the processor core, independent of the voltage supplied to the I/O buffers on the V<sub>CC3</sub> pins. Upon sampling VCC2DET Low to identify dual-voltage processor requirements, system logic should sample VCC2H/L# to identify the core voltage requirements for 2.9V and 3.2V products (High) and 2.2V products (Low).

**Driven**

VCC2H/L# always equals 0 and is never floated for 2.2V products—even during the Tri-State Test mode. To ensure proper operation for 2.9V and 3.2V products, system logic that samples VCC2H/L# should design a weak pullup resistor for this signal.

**Table 38. Output Pin Float Conditions**

Name	Floated At:	Note
VCC2DET	Always Driven	*
VCC2H/L#	Always Driven	*
<b>Note:</b> * All outputs except VCC2DET, VCC2H/L#, and TDO float during the Tri-State Test mode.		

## **5** *Signal Descriptions*

---

### **W/R# (Write/Read)**

---

#### **Output**

#### ***Summary***

The processor drives W/R# to indicate whether it is performing a write or a read cycle on the bus. In addition, W/R# is used to define other bus cycles, including interrupt acknowledge and special cycles. See Table 44 on page 186 for more details.

#### ***Driven and Floated***

W/R# is driven off the same clock edge as ADS# and remains in the same state until the clock edge on which NA# or the last expected BRDY# of the cycle is sampled asserted. W/R# is driven during memory cycles, I/O cycles, special bus cycles, and interrupt acknowledge cycles.

W/R# is floated off the clock edge that BOFF# is sampled asserted and off the clock edge that the processor asserts HLDA in response to HOLD.

**WB/WT# (Writeback or Writethrough)****Input****Summary**

WB/WT#, together with PWT, specifies the data cache-line state during cacheable read misses and write hits to shared cache lines.

If WB/WT# = 0 or PWT = 1 during a cacheable read miss or write hit to a shared cache line, the accessed line is cached in the shared state. This is referred to as the writethrough state because all write cycles to this cache line are driven externally on the bus.

If WB/WT# = 1 and PWT = 0 during a cacheable read miss or a write hit to a shared cache line, the accessed line is cached in the exclusive state. Subsequent write hits to the same line cause its state to transition from exclusive to modified. This is referred to as the writeback state because the data cache can contain modified cache lines that are subject to be written back—referred to as a writeback cycle—as the result of an inquire cycle, an internal snoop, a flush operation, or the WBINVD instruction.

**Sampled**

WB/WT# is sampled on the clock-edge that the first BRDY# or NA# of a bus cycle is sampled asserted. If the cycle is a burst read, WB/WT# is ignored during the last three assertions of BRDY#. WB/WT# is sampled during memory read and non-writeback write cycles and is ignored during all other types of cycles.

# 5 Signal Descriptions

**Table 39. Input Pin Types**

Name	Type	Note	Name	Type	Note
A20M#	Asynchronous	1	IGNNE#	Asynchronous	1
AHOLD	Synchronous		INIT	Asynchronous	2
BF[2:0]	Synchronous	4	INTR	Asynchronous	1
BOFF#	Synchronous		INV	Synchronous	
BRDY#	Synchronous		KEN#	Synchronous	
BRDYC#	Synchronous	7	NA#	Synchronous	
CLK	Clock		NMI	Asynchronous	2
EADS#	Synchronous		RESET	Asynchronous	5, 6
EWBE#	Synchronous		SMI#	Asynchronous	2
FLUSH#	Asynchronous	2, 3	STPCLK#	Asynchronous	1
HOLD	Synchronous		WB/WT#	Synchronous	

**Notes:**

1. These level-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must be asserted for a minimum pulse width of two clocks.
2. These edge-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must have been negated at least two clocks prior to assertion and must remain asserted at least two clocks.
3. FLUSH# is also sampled during the falling transition of RESET and can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met relative to the clock edge before the clock edge on which RESET is sampled negated. If asserted asynchronously, FLUSH# must meet a minimum setup and hold time of two clocks relative to the negation of RESET.
4. BF[2:0] are sampled during the falling transition of RESET. They must meet a minimum setup time of 1.0 ns and a minimum hold time of two clocks relative to the negation of RESET.
5. During the initial power-on reset of the processor, RESET must remain asserted for a minimum of 1.0 ms after CLK and V<sub>CC</sub> reach specification before it is negated.
6. During a warm reset, while CLK and V<sub>CC</sub> are within their specification, RESET must remain asserted for a minimum of 15 clocks prior to its negation.
7. BRDYC# is also sampled during the falling transition of RESET. If RESET is driven synchronously, BRDYC# must meet the specified hold time relative to the negation of RESET. If asserted asynchronously, BRDYC# must meet a minimum setup and hold time of two clocks relative to the negation of RESET.

Table 40. Output Pin Float Conditions

Name	Floated At: (Note 1)	Note	Name	Floated At: (Note 1)	Note
A[4:3]	HLDA, AHOLD, BOFF#	2, 3	HLDA	Always Driven	
ADS#	HLDA, BOFF#	2	LOCK#	HLDA, BOFF#	2
ADSC#	HLDA, BOFF#	2	M/IO#	HLDA, BOFF#	2
APCHK#	Always Driven		PCD	HLDA, BOFF#	2
BE[7:0]#	HLDA, BOFF#	2	PCHK#	Always Driven	
BREQ	Always Driven		PWT	HLDA, BOFF#	2
CACHE#	HLDA, BOFF#	2	SCYC	HLDA, BOFF#	2
D/C#	HLDA, BOFF#	2	SMIACK#	Always Driven	
FERR#	Always Driven		VCC2DET	Always Driven	
HIT#	Always Driven		VCC2H/L#	Always Driven	
HITM#	Always Driven		W/R#	HLDA, BOFF#	2

**Notes:**

1. All outputs except VCC2DET, VCC2H/L#, and TDO float during the Tri-State Test mode.
2. Floated off the clock edge that BOFF# is sampled asserted and off the clock edge that HLDA is asserted.
3. Floated off the clock edge that AHOLD is sampled asserted.

Table 41. Input/Output Pin Float Conditions

Name	Floated At: (Note 1)	Note
A[31:5]	HLDA, AHOLD, BOFF#	2,3
AP	HLDA, AHOLD, BOFF#	2,3
D[63:0]	HLDA, BOFF#	2
DP[7:0]	HLDA, BOFF#	2

**Notes:**

1. All outputs except VCC2DET and TDO float during the Tri-State Test mode.
2. Floated off the clock edge that BOFF# is sampled asserted and off the clock edge that HLDA is asserted.
3. Floated off the clock edge that AHOLD is sampled asserted.

Table 42. Test Pins

Name	Type	Note
TCK	Clock	
TDI	Input	Sampled on the rising edge of TCK
TDO	Output	Driven on the falling edge of TCK
TMS	Input	Sampled on the rising edge of TCK
TRST#	Input	Asynchronous (Independent of TCK)

# 5 Signal Descriptions

**Table 43. Bus Cycle Definition**

Bus Cycle Initiated	Generated by the Processor				Generated by the System
	M/IO#	D/C#	W/R#	CACHE#	KEN#
Code Read, Instruction Cache Line Fill	1	0	0	0	0
Code Read, Noncacheable	1	0	0	1	x
Code Read, Noncacheable	1	0	0	x	1
Encoding for Special Cycle	0	0	1	1	x
Interrupt Acknowledge	0	0	0	1	x
I/O Read	0	1	0	1	x
I/O Write	0	1	1	1	x
Memory Read, Data Cache Line Fill	1	1	0	0	0
Memory Read, Noncacheable	1	1	0	1	x
Memory Read, Noncacheable	1	1	0	x	1
Memory Write, Data Cache Writeback	1	1	1	0	x
Memory Write, Noncacheable	1	1	1	1	x
<b>Note:</b> <i>x means "don't care"</i>					

**Table 44. Special Cycles**

Special Cycle	A4	BE7#	BE6#	BE5#	BE4#	BE3#	BE2#	BE1#	BE0#	M/IO#	D/C#	W/R#	CACHE#	KEN#
Stop Grant	1	1	1	1	1	1	0	1	1	0	0	1	1	x
Flush Acknowledge (FLUSH# sampled asserted)	0	1	1	1	0	1	1	1	1	0	0	1	1	x
Writeback (WBINVD instruction)	0	1	1	1	1	0	1	1	1	0	0	1	1	x
Halt	0	1	1	1	1	1	0	1	1	0	0	1	1	x
Flush (INVD, WBINVD instruction)	0	1	1	1	1	1	1	0	1	0	0	1	1	x
Shutdown	0	1	1	1	1	1	1	1	0	0	0	1	1	x
<b>Note:</b> <i>x means "don't care"</i>														

# 6

## Bus Cycles

The following sections describe and illustrate the timing and relationship of bus signals during various types of bus cycles. A representative set of bus cycles is illustrated.

### Timing Diagrams

---

The timing diagrams illustrate the signals on the external local bus as a function of time, as measured by the bus clock (CLK). Throughout this chapter, the term *clock* refers to a single bus-clock cycle. A clock extends from one rising CLK edge to the next rising CLK edge. The processor samples and drives most signals relative to the rising edge of CLK. The exceptions to this rule include the following:

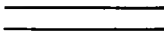





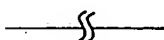
- BF[2:0]—Sampled on the falling edge of RESET
- FLUSH#, BRDYC#—Sampled on the falling edge of RESET, also sampled on the rising edge of CLK
- All inputs and outputs are sampled relative to TCK in Boundary-Scan Test Mode. Inputs are sampled on the rising edge of TCK, outputs are driven off of the falling edge of TCK.

For each signal in the timing diagrams, the High level represents 1, the Low level represents 0, and the Middle level represents the floating (high-impedance) state. When both the



## 6 Bus Cycles

High and Low levels are shown, the meaning depends on the signal. A single signal indicates 'don't care'. In the case of bus activity, if both High and Low levels are shown, it indicates the processor, alternate master, or system logic is driving a value, but this value may or may not be valid. (For example, the value on the address bus is valid only during the assertion of ADS#, but addresses are also driven on the bus at other times.) Figure 52 defines the different waveform representations.

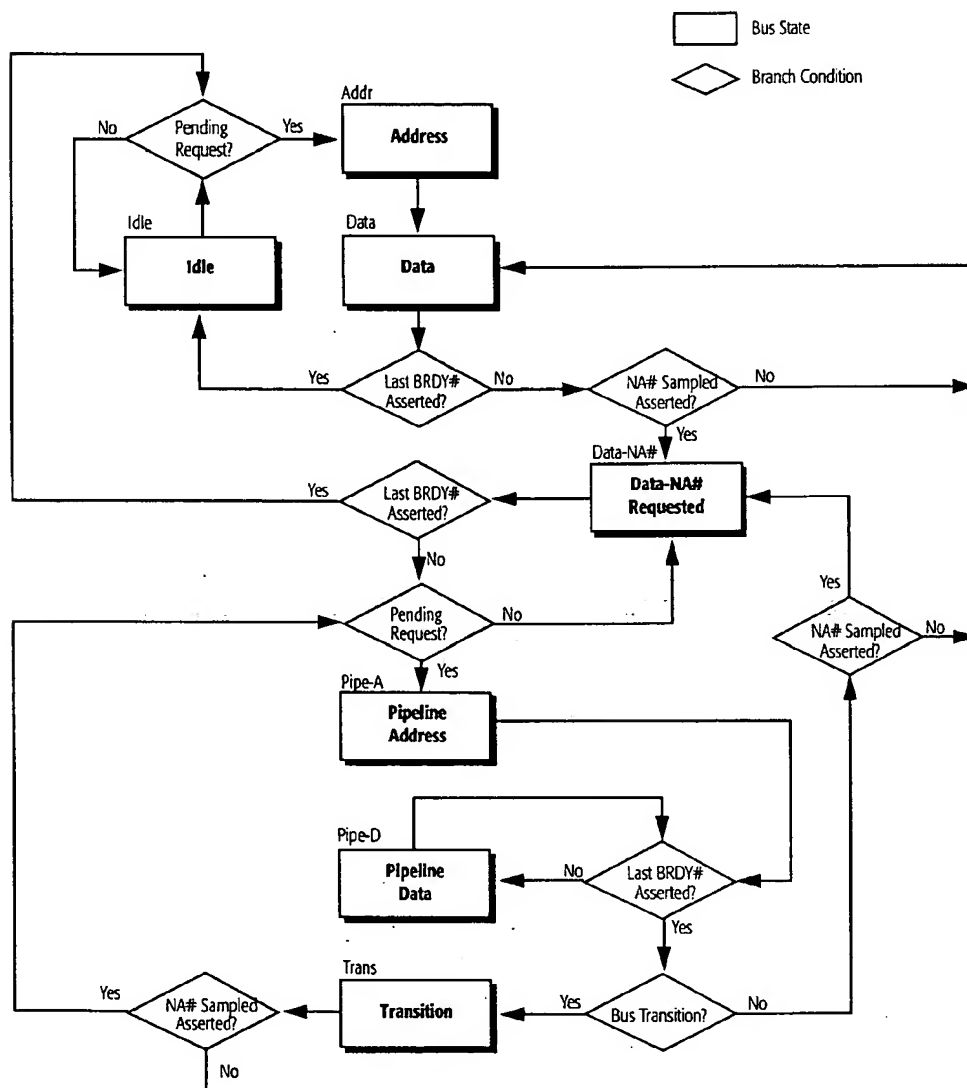
Waveform	Description
	Don't care or bus is driven
	Signal or bus is changing from Low to High
	Signal or bus is changing from High to Low
	Bus is changing
	Bus is changing from valid to invalid
	Signal or bus is floating
	Denotes multiple clock periods

**Figure 52. Waveform Definitions**

For all active-High signals, the term *asserted* means the signal is in the High-voltage state and the term *negated* means the signal is in the Low-voltage state. For all active-Low signals, the term *asserted* means the signal is in the Low-voltage state and the term *negated* means the signal is in the High-voltage state.

See "Signal Terminology" on page 139 for definitions of terms related to signals.

## Bus State Machine Diagram



**Note:** The processor transitions to the IDLE state on the clock edge on which BOFF# or RESET is sampled asserted.

**Figure 53. Bus State Machine Diagram**

## **6** *Bus Cycles*

---

### **Idle**

The processor does not drive the system bus in the Idle state and remains in this state until a new bus cycle is requested. The processor enters this state off the clock edge on which the last BRDY# of a cycle is sampled asserted during the following conditions:

- The processor is in the Data state
- The processor is in the Data-NA# Requested state and no internal pending cycle is requested

In addition, the processor is forced into this state when the system logic asserts RESET or BOFF#. The transition to this state occurs on the clock edge on which RESET or BOFF# is sampled asserted.

### **Address**

In this state, the processor drives ADS# to indicate the beginning of a new bus cycle by validating the address and control signals. The processor remains in this state for one clock and unconditionally enters the Data state on the next clock edge.

### **Data**

In the Data state, the processor drives the data bus during a write cycle or expects data to be returned during a read cycle. The processor remains in this state until either NA# or the last BRDY# is sampled asserted. If the last BRDY# is sampled asserted or both the last BRDY# and NA# are sampled asserted on the same clock edge, the processor enters the Idle state. If NA# is sampled asserted first, the processor enters the Data-NA# Requested state.

### **Data-NA# Requested**

If the processor samples NA# asserted while in the Data state and the current bus cycle is not completed (the last BRDY# is not sampled asserted), it enters the Data-NA# Requested state. The processor remains in this state until either the last BRDY#

is sampled asserted or an internal pending cycle is requested. If the last BRDY# is sampled asserted before the processor drives a new bus cycle, the processor enters the Idle state (no internal pending cycle is requested) or the Address state (processor has a internal pending cycle).

## **Pipeline Address**

In this state, the processor drives ADS# to indicate the beginning of a new bus cycle by validating the address and control signals. In this state, the processor is still waiting for the current bus cycle to be completed (until the last BRDY# is sampled asserted). If the last BRDY# is not sampled asserted, the processor enters the Pipeline Data state.

If the processor samples the last BRDY# asserted in this state, it determines if a bus transition is required between the current bus cycle and the pipelined bus cycle. A bus transition is required when the data bus direction changes between bus cycles, such as a memory write cycle followed by a memory read cycle. If a bus transition is required, the processor enters the Transition state for one clock to prevent data bus contention. If a bus transition is not required, the processor enters the Data state.

The processor does not transition to the Data-NA# Requested state from the Pipeline Address state because the processor does not begin sampling NA# until it has exited the Pipeline Address state.

## **Pipeline Data**

Two bus cycles are concurrently executing in this state. The processor cannot issue any additional bus cycles until the current bus cycle is completed. The processor drives the data bus during write cycles or expects data to be returned during read cycles for the current bus cycle until the last BRDY# of the current bus cycle is sampled asserted.

If the processor samples the last BRDY# asserted in this state, it determines if a bus transition is required between the current bus cycle and the pipelined bus cycle. If the bus transition is required, the processor enters the Transition state for one clock

## **6** *Bus Cycles*

---

to prevent data bus contention. If a bus transition is not required, the processor enters the Data state (NA# was not sampled asserted) or the Data-NA# Requested state (NA# was sampled asserted).

### **Transition**

The processor enters this state for one clock during data bus transitions and enters the Data state on the next clock edge if NA# is not sampled asserted. The sole purpose of this state is to avoid bus contention caused by bus transitions during pipeline operation.

### **Memory Reads and Writes**

---

The processor performs single or burst memory bus cycles. The single-transfer memory bus cycle transfers 1, 2, 4, or 8 bytes and requires a minimum of two clocks. Misaligned instructions or operands result in a split cycle, which requires multiple transactions on the bus. A burst cycle consists of four back-to-back 8-byte (64-bit) transfers on the data bus.

### **Single-Transfer Memory Read and Write**

Figure 54 on page 193 shows a single-transfer read from memory, followed by two single-transfer writes to memory. For the memory read cycle, the processor asserts ADS# for one clock to validate the bus cycle and also drives A[31:3], BE[7:0]#, D/C#, W/R#, and M/IO# to the bus. The processor then waits for the system logic to return the data on D[63:0] (with DP[7:0] for parity checking) and assert BRDY#. The processor samples BRDY# on every clock edge starting with the clock edge after the clock edge that negates ADS#. See “BRDY# (Burst Ready)” on page 149.

During the read cycle, the processor drives PCD, PWT, and CACHE# to indicate its caching and cache-coherency intent for the access. The system logic returns KEN# and WB/WT# to either confirm or change this intent. If the processor asserts PCD and negates CACHE#, the accesses are noncacheable, even though the system logic asserts KEN# during the BRDY# to indicate its support for cacheability. The processor (which drives CACHE#) and the system logic (which drives KEN#) must agree in order for an access to be cacheable.

## Bus Cycles 6

The processor can drive another cycle (in this example, a write cycle) by asserting ADS# off the next clock edge after BRDY# is sampled asserted. Therefore, an idle clock is guaranteed between any two bus cycles. The processor drives D[63:0] with valid data one clock edge after the clock edge on which ADS# is asserted. To minimize processor idle times, the system logic stores the address and data in write buffers, returns BRDY#, and performs the store to memory later. If the processor samples EWBE# negated during a write cycle, it suspends certain activities until EWBE# is sampled asserted. See “EWBE# (External Write Buffer Empty)” on page 157. In Figure 54, the second write cycle occurs during the execution of a serializing instruction. The processor delays the following cycle until EWBE# is sampled asserted.

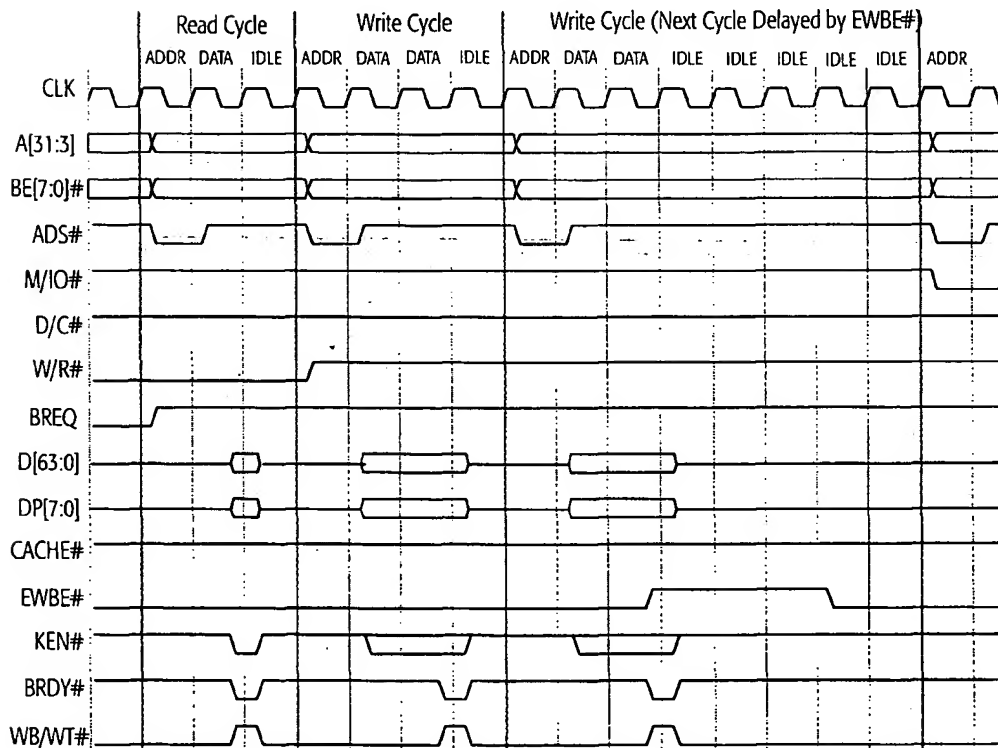


Figure 54. Non-Pipelined Single-Transfer Memory Read/Write and Write Delayed by EWBE#

## 6 Bus Cycles

### Misaligned Single-Transfer Memory Read and Write

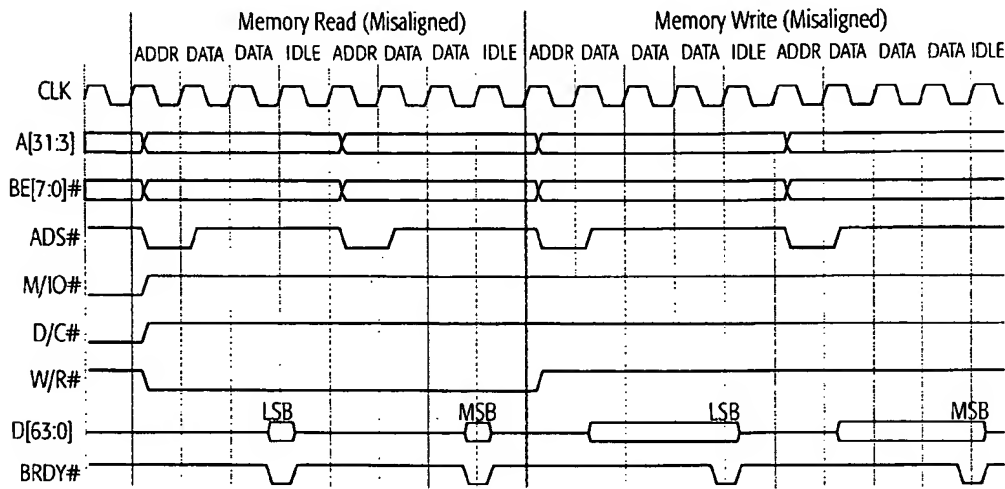
Figure 55 on page 195 shows a misaligned (split) memory read followed by a misaligned memory write. Any cycle that is not aligned as defined in “SCYC (Split Cycle)” on page 175 is considered misaligned. When the processor encounters a misaligned access, it determines the appropriate pair of bus cycles—each with its own ADS# and BRDY#—required to complete the access.

The processor performs misaligned memory reads and memory writes using least-significant bytes (LSBs) first followed by most-significant bytes (MSBs). Table 45 shows the order. In the first memory read cycle in Figure 55, the processor reads the least-significant bytes. Immediately after the processor samples BRDY# asserted, it drives the second bus cycle to read the most-significant bytes to complete the misaligned transfer.

**Table 45. Bus-Cycle Order During Misaligned Transfers**

Type of Access	First Cycle	Second Cycle
Memory Read	LSBs	MSBs
Memory Write	LSBs	MSBs

Similarly, the misaligned memory write cycle in Figure 55 transfers the LSBs to the memory bus first. In the next cycle, after the processor samples BRDY# asserted, the MSBs are written to the memory bus.



**Figure 55. Misaligned Single-Transfer Memory Read and Write**

## Burst Reads and Pipelined Burst Reads

Figure 56 on page 197 shows normal burst read cycles and a pipelined burst read cycle. The AMD-K6 3D processor drives CACHE# and ADS# together to specify that the current bus cycle is a burst cycle. If the processor samples KEN# asserted with the first BRDY#, it performs burst transfers. During the burst transfers, the system logic must ignore BE[7:0]# and must return all eight bytes beginning at the starting address the processor asserts on A[31:3]. Depending on the starting address, the system logic must determine the successive quadword addresses (A[4:3]) for each transfer in a burst, as shown in Table 46 on page 196. The processor expects the second, third, and fourth quadwords to occur in the sequences shown in Table 46.



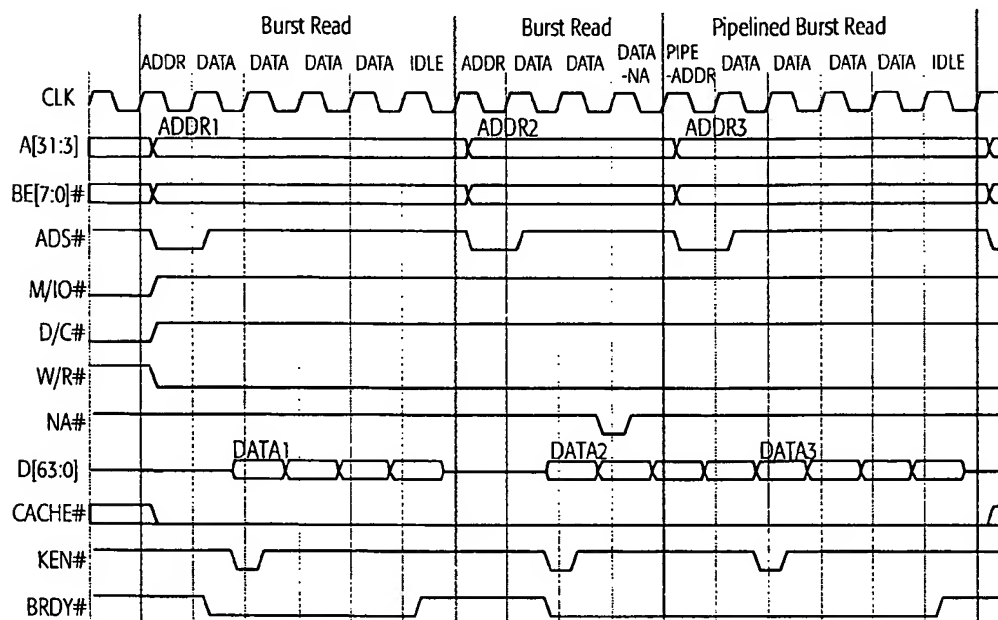
## 6 Bus Cycles

**Table 46. A[4:3] Address-Generation Sequence During Bursts**

Address Driven By Processor on A[4:3]	A[4:3] Addresses of Subsequent Quadwords* Generated By System Logic		
	Quadword 2	Quadword 3	Quadword 4
Quadword 1			
00b	01b	10b	11b
01b	00b	11b	10b
10b	11b	00b	01b
11b	10b	01b	00b
<b>Note:</b> * quadword = 8 bytes			

In Figure 56 on page 197, the processor drives **CACHE#** throughout all burst read cycles. In the first burst read cycle, the processor drives **ADS#** and **CACHE#**, then samples **BRDY#** on every clock edge starting with the clock edge after the clock edge that negates **ADS#**. The processor samples **KEN#** asserted on the clock edge on which the first **BRDY#** is sampled asserted, executes a 32-byte burst read cycle, and expects a total of four **BRDY#** signals. An ideal no-wait state access is shown in Figure 56, whereas most system logic solutions add wait states between the transfers.

The second burst read cycle illustrates a similar sequence, but the processor samples **NA#** asserted on the same clock edge that the first **BRDY#** is sampled asserted. **NA#** assertion indicates the system logic is requesting the processor to output the next address early (also known as a pipeline transfer request). Without waiting for the current cycle to complete, the processor drives **ADS#** and related signals for the next burst cycle. Pipelining can reduce processor cycle-to-cycle idle times.



**Figure 56. Burst Reads and Pipelined Burst Reads**

## Burst Writeback

Figure 57 on page 198 shows a burst read followed by a writeback transaction. The processor initiates writebacks under the following conditions:

- **Replacement**—If a cache-line fill is initiated for a cache line currently filled with valid entries, the processor uses a least-recently-allocated (LRA) algorithm to select a line for replacement. Before a replacement is made to a data cache line that is in the modified state, the modified line is scheduled to be written back to memory.
- **Internal Snoop**—The processor snoops the data cache whenever an instruction-cache line is read, and it snoops the instruction cache whenever a data cache line is written. This snooping is performed to determine whether the same address is stored in both caches, a situation that is taken to imply the occurrence of self-modifying code. If a snoop hits a data cache line in the modified state, the line is written back to memory before being invalidated.

## 6 Bus Cycles

- **WBINVD Instruction**—When the processor executes a WBINVD instruction, it writes back all modified lines in the data cache and then invalidates all lines in both caches.
- **Cache Flush**—When the processor samples FLUSH# asserted, it executes a flush acknowledge special cycle and writes back all modified lines in the data cache and then invalidates all lines in both caches.

The processor drives writeback cycles during inquire or cache flush cycles. The writeback shown in Figure 57 is caused by a cache-line replacement. The processor completes the burst read cycle that fills the cache line. Immediately following the burst read cycle is the burst writeback cycle that represents the modified line to be written back to memory. D[63:0] are driven one clock edge after the clock edge on which ADS# is asserted and are subsequently changed off the clock edge on which each of the four BRDY# signals of the burst cycle are sampled asserted.

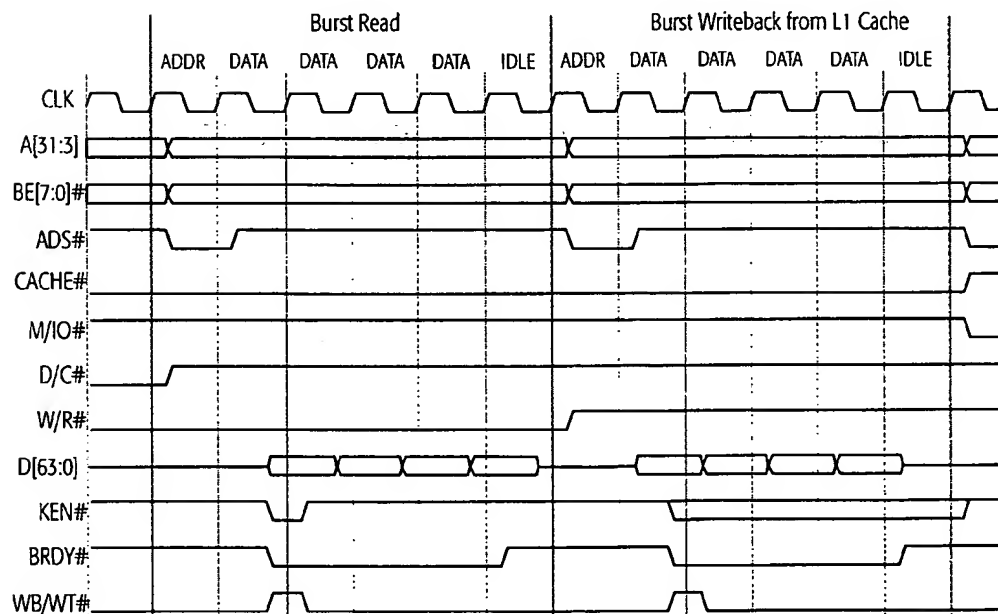


Figure 57. Burst Writeback due to Cache-Line Replacement

## **I/O Read and Write**

---

### **Basic I/O Read and Write**

The AMD-K6 3D processor accesses I/O when it executes an I/O instruction (for example, IN or OUT). Figure 58 on page 200 shows an I/O read followed by an I/O write. The processor drives M/IO# Low and D/C# High during I/O cycles. In this example, the first cycle shows a single wait state I/O read cycle. It follows the same sequence as a single-transfer memory read cycle. The processor drives ADS# to initiate the bus cycle, then it samples BRDY# on every clock edge starting with the clock edge after the clock edge that negates ADS#. The system logic must return BRDY# to complete the cycle. When the processor samples BRDY# asserted, it can assert ADS# for the next cycle off the next clock edge. (In this example, an I/O write cycle.)

The I/O write cycle is similar to a memory write cycle, but the processor drives M/IO# low during an I/O write cycle. The processor asserts ADS# to initiate the bus cycle. The processor drives D[63:0] with valid data one clock edge after the clock edge on which ADS# is asserted. The system logic must assert BRDY# when the data is properly stored to the I/O destination. The processor samples BRDY# on every clock edge starting with the clock edge after the clock edge that negates ADS#. In this example, two wait states are inserted while the processor waits for BRDY# to be asserted.

## 6 Bus Cycles

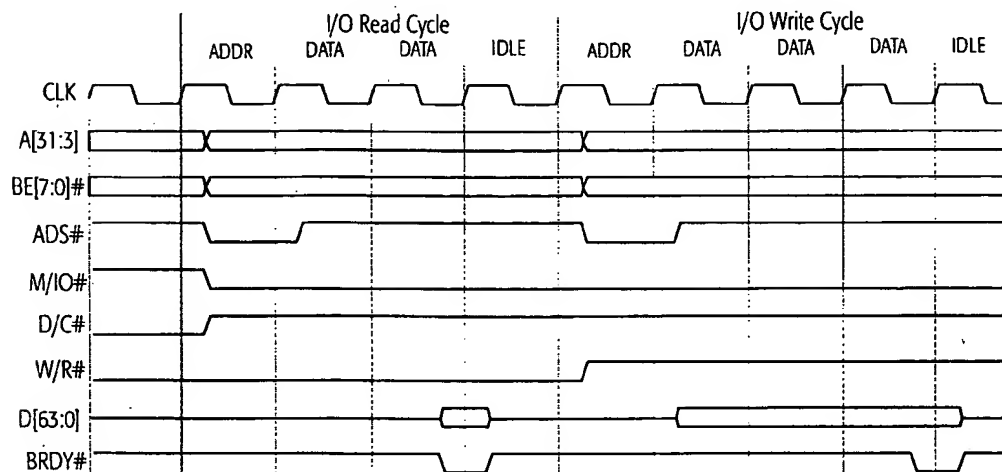


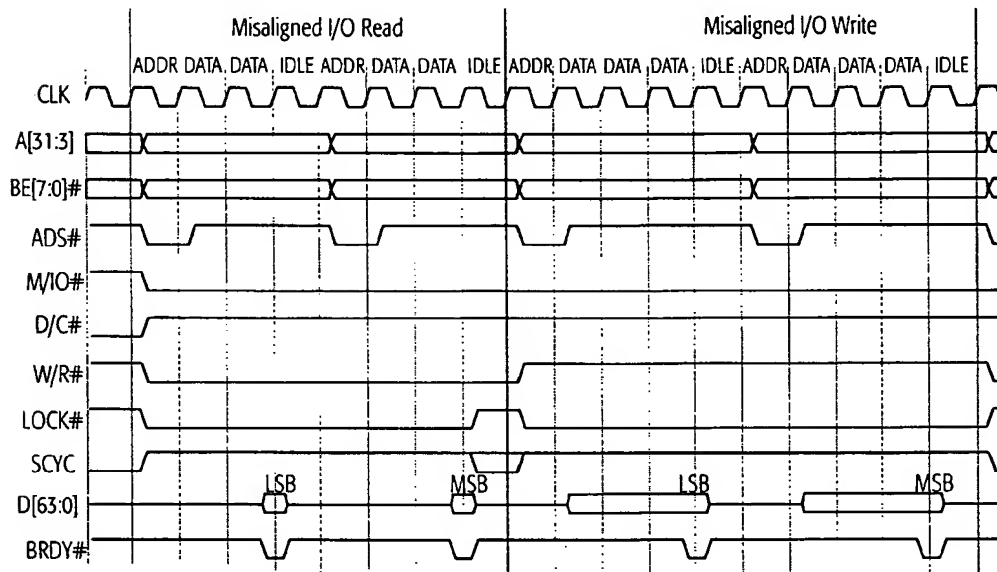
Figure 58. Basic I/O Read and Write

### Misaligned I/O Read and Write

Table 47 shows the misaligned I/O read and write cycle order executed by the processor. In Figure 59 on page 201, the least-significant bytes (LSBs) are transferred first. Immediately after the processor samples BRDY# asserted, it drives the second bus cycle to transfer the most-significant bytes (MSBs) to complete the misaligned bus cycle.

Table 47. Bus-Cycle Order During Misaligned I/O Transfers

Type of Access	First Cycle	Second Cycle
I/O Read	LSBs	MSBs
I/O Write	LSBs	MSBs



**Figure 59. Misaligned I/O Transfer**

## Inquire and Bus Arbitration Cycles

The processor provides built-in level-one data and instruction caches. Each cache is 32 Kbytes and two-way set-associative. The system logic or other bus master devices can initiate an inquire cycle to maintain cache/memory coherency. In response to the inquire cycle, the processor compares the inquire address with its cache tag addresses in both caches, and, if necessary, updates the MESI state of the cache line and performs writebacks to memory.

An inquire cycle can be initiated by asserting AHOLD, BOFF#, or HOLD. AHOLD is exclusively used to support inquire cycles. During AHOLD-initiated inquire cycles, the processor only floats the address bus. BOFF# provides the fastest access to the bus because it aborts any processor cycle that is in-progress, whereas AHOLD and HOLD both permit an in-progress bus cycle to complete. During HOLD-initiated and BOFF#-initiated inquire cycles, the processor floats all of its bus-driving signals.

## **6** *Bus Cycles*

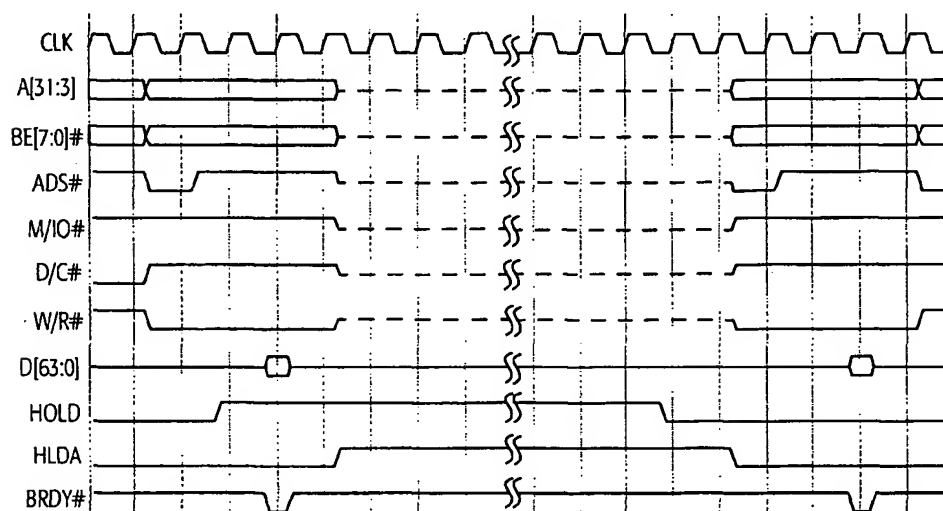
### **Hold and Hold Acknowledge Cycle**

The system logic or another bus device can assert HOLD to initiate an inquire cycle or to gain full control of the bus. When the processor samples HOLD asserted, it completes any in-progress bus cycle and asserts HLDA to acknowledge release of the bus. The processor floats the following signals off the same clock edge that HLDA is asserted:

- |            |           |
|------------|-----------|
| ■ A[31:3]  | ■ DP[7:0] |
| ■ ADS#     | ■ LOCK#   |
| ■ AP#      | ■ M/IO#   |
| ■ BE[7:0]# | ■ PCD     |
| ■ CACHE#   | ■ PWT     |
| ■ D[63:0]  | ■ SCYC    |
| ■ D/C#     | ■ W/R#    |

Figure 60 on page 203 shows a basic HOLD/HLDA operation. In this example, the processor samples HOLD asserted during the memory read cycle. It continues the current memory read cycle until BRDY# is sampled asserted. The processor drives HLDA and floats its outputs one clock edge after the last BRDY# of the cycle is sampled asserted. The system logic can assert HOLD for as long as it needs to utilize the bus. The processor samples HOLD on every clock edge but does not assert HLDA until any in-progress cycle or sequence of locked cycles is completed.

When the processor samples HOLD negated during a hold acknowledge cycle, it negates HLDA off the next clock edge. The processor regains control of the bus and can assert ADS# off the same clock edge on which HLDA is negated.



**Figure 60. Basic HOLD/HLDA Operation**

## **HOLD-Initiated Inquire Hit to Shared or Exclusive Line**

Figure 61 on page 204 shows a HOLD-initiated inquire cycle. In this example, the processor samples HOLD asserted during the burst memory read cycle. The processor completes the current cycle (until the last expected BRDY# is sampled asserted), asserts HLDA and floats its outputs as described on page 202.

The system logic drives an inquire cycle within the hold acknowledge cycle. It asserts EADS#, which validates the inquire address on A[31:5]. If EADS# is sampled asserted before HOLD is sampled negated, the processor recognizes it as a valid inquire cycle.

In Figure 61, the processor asserts HIT# and negates HITM# on the clock edge after the clock edge on which EADS# is sampled asserted, indicating the current inquire cycle hit a shared or exclusive cache line. (Shared and exclusive cache lines in the processor data or instruction cache have the same contents as the data in the external memory.) During an inquire cycle, the processor samples INV to determine whether the addressed



## 6 Bus Cycles

cache line found in the processor's instruction or data cache transitions to the invalid state or the shared state. In this example, the processor samples INV asserted with EADS#, which invalidates the cache line.

The system logic can negate HOLD off the same clock edge on which EADS# is sampled asserted. The processor continues driving HIT# in the same state until the next inquire cycle. HITM# is not asserted unless HIT# is asserted.

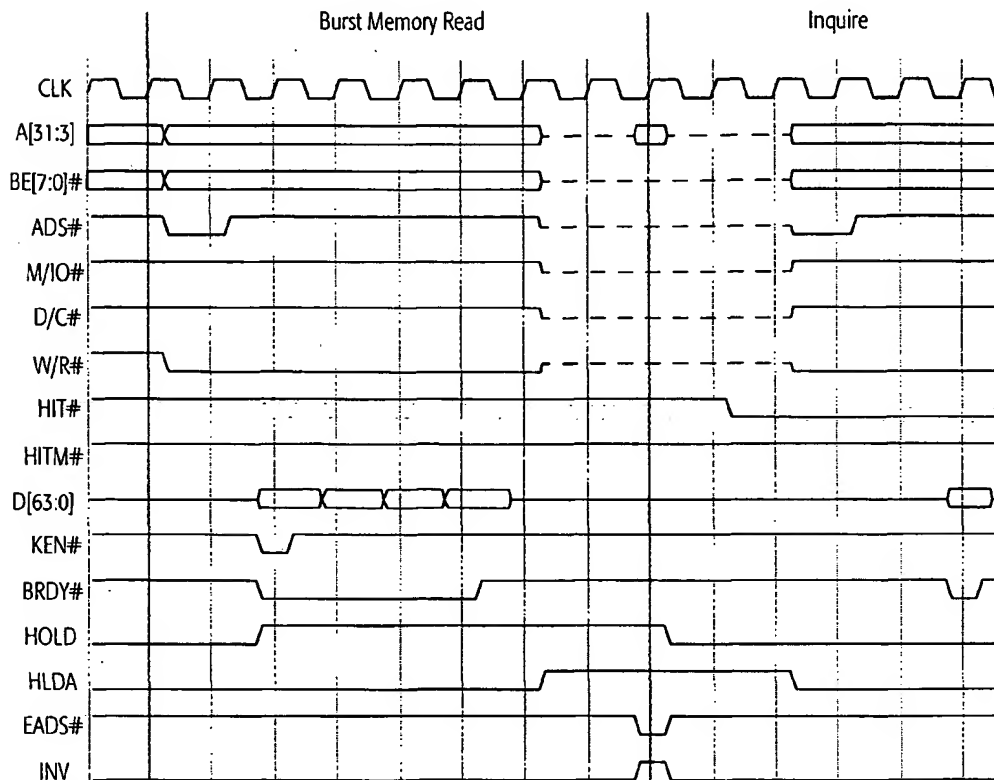


Figure 61. HOLD-Initiated Inquire Hit to Shared or Exclusive Line

**HOLD-Initiated Inquire Hit to Modified Line**

Figure 62 on page 206 shows the same sequence as Figure 61 on page 204, but in Figure 62 the inquire cycle hits a modified line and the processor asserts both HIT# and HITM#. In this example, the processor performs a writeback cycle immediately after the inquire cycle. It updates the modified cache line to the external memory (normally, level-two cache or DRAM). The processor uses the address (A[31:5]) that was latched during the inquire cycle to perform the writeback cycle. The processor asserts HITM# throughout the writeback cycle and negates HITM# one clock edge after the last expected BRDY# of the writeback is sampled asserted.

When the processor samples EADS# during the inquire cycle, it also samples INV to determine the cache line MESI state after the inquire cycle. If INV is sampled asserted during an inquire cycle, the processor transitions the line (if found) to the invalid state, regardless of its previous state. The cache line invalidation operation is not visible on the bus. If INV is sampled negated during an inquire cycle, the processor transitions the line (if found) to the shared state. In Figure 62 the processor samples INV asserted during the inquire cycle.

In a HOLD-initiated inquire cycle, the system logic can negate HOLD off the same clock edge on which EADS# is sampled asserted. The processor drives HIT# and HITM# on the clock edge after the clock edge on which EADS# is sampled asserted.

## 6 Bus Cycles

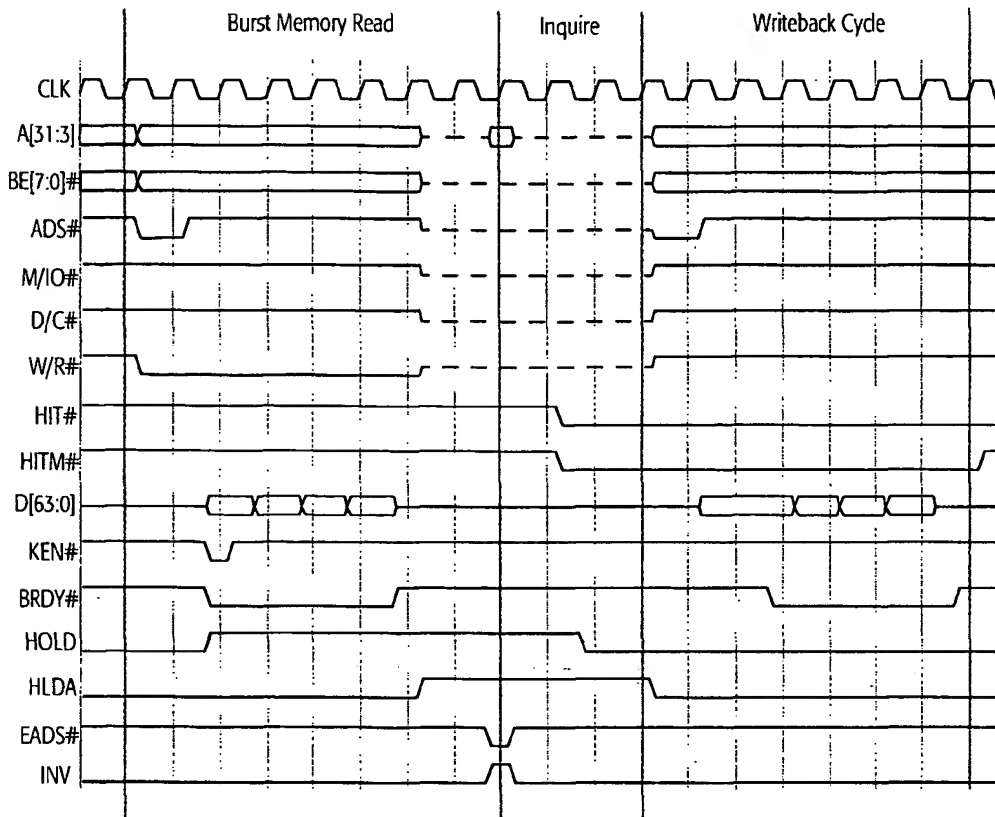


Figure 62. HOLD-Initiated Inquire Hit to Modified Line

## **AHOLD-Initiated Inquire Miss**

AHOLD can be asserted by the system to initiate one or more inquire cycles. To allow the system to drive the address bus during an inquire cycle, the processor floats A[31:3] and AP off the clock edge on which AHOLD is sampled asserted. The data bus and all other control and status signals remain under the control of the processor and are not floated. This functionality allows a bus cycle in progress when AHOLD is sampled asserted to continue to completion. The processor resumes driving the address bus off the clock edge on which AHOLD is sampled negated.

In Figure 63 on page 208, the processor samples AHOLD asserted during the memory burst read cycle, and it floats the address bus off the same clock edge on which it samples AHOLD asserted. While the processor still controls the bus, it completes the current cycle until the last expected BRDY# is sampled asserted. The system logic drives EADS# with an inquire address on A[31:5] during an inquire cycle. The processor samples EADS# asserted and compares the inquire address to its tag address in both the instruction and data caches. In Figure 63, the inquire address misses the tag address in the processor (both HIT# and HITM# are negated). Therefore, the processor proceeds to the next cycle when it samples AHOLD negated. (The processor can drive a new cycle by asserting ADS# off the same clock edge that it samples AHOLD negated.)

For an AHOLD-initiated inquire cycle to be recognized, the processor must sample AHOLD asserted for at least two consecutive clocks before it samples EADS# asserted. If the processor detects an address parity error during an inquire cycle, APCHK# is asserted for one clock. The system logic must respond appropriately to the assertion of this signal.

## 6 Bus Cycles

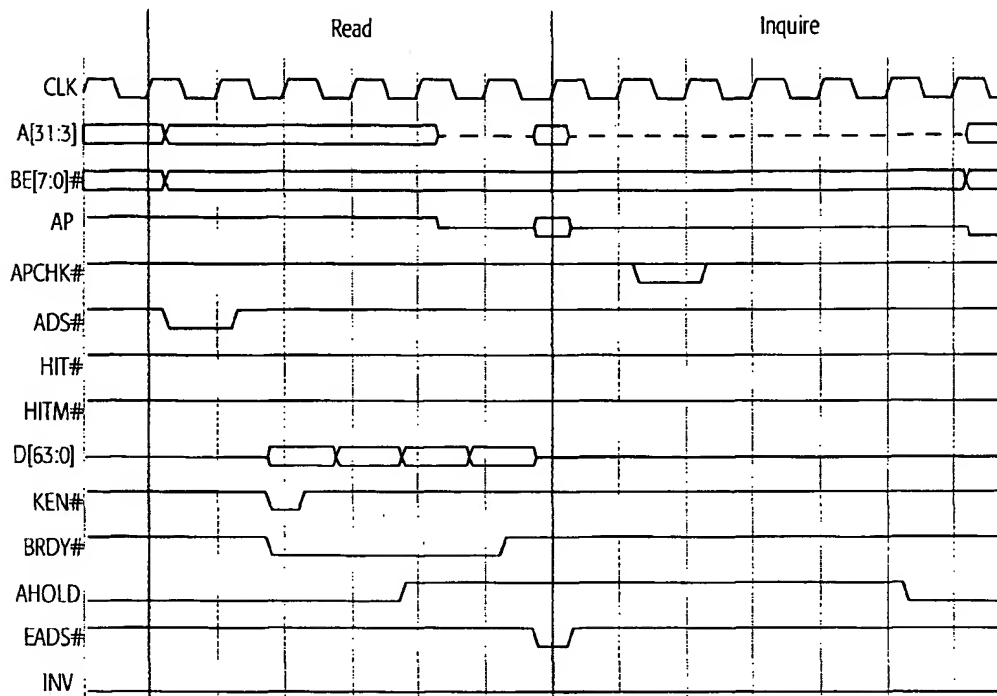


Figure 63. AHOLD-Initiated Inquire Miss

## AHOLD-Initiated Inquire Hit to Shared or Exclusive Line

In Figure 64, the processor asserts HIT# and negates HITM# off the clock edge after the clock edge on which EADS# is sampled asserted, indicating the current inquire cycle hits either a shared or exclusive line. (HIT# is driven in the same state until the next inquire cycle.) The processor samples INV asserted during the inquire cycle and transitions the line to the invalid state regardless of its previous state.

During an AHOLD-initiated inquire cycle, the processor samples AHOLD on every clock edge until it is negated. In Figure 64, the processor asserts ADS# off the same clock on which AHOLD is sampled negated. If the inquire cycle hits a modified line, the processor performs a writeback cycle before it drives a new bus cycle. The next section describes the AHOLD-initiated inquire cycle that hits a modified line.

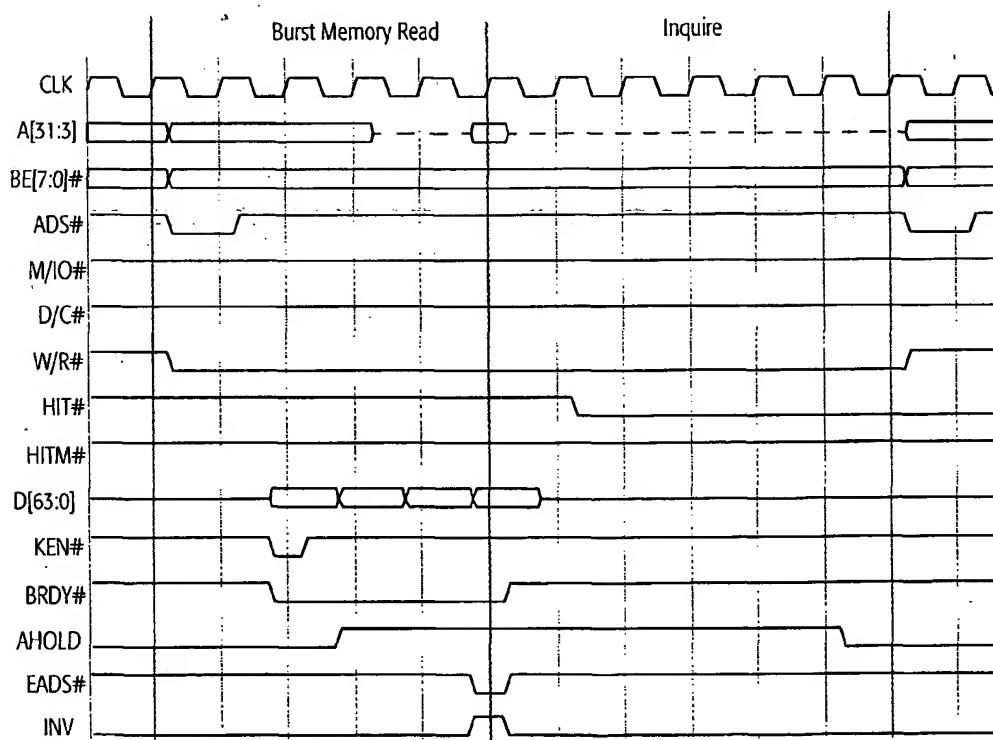


Figure 64. AHOLD-Initiated Inquire Hit to Shared or Exclusive Line

## **6** *Bus Cycles*

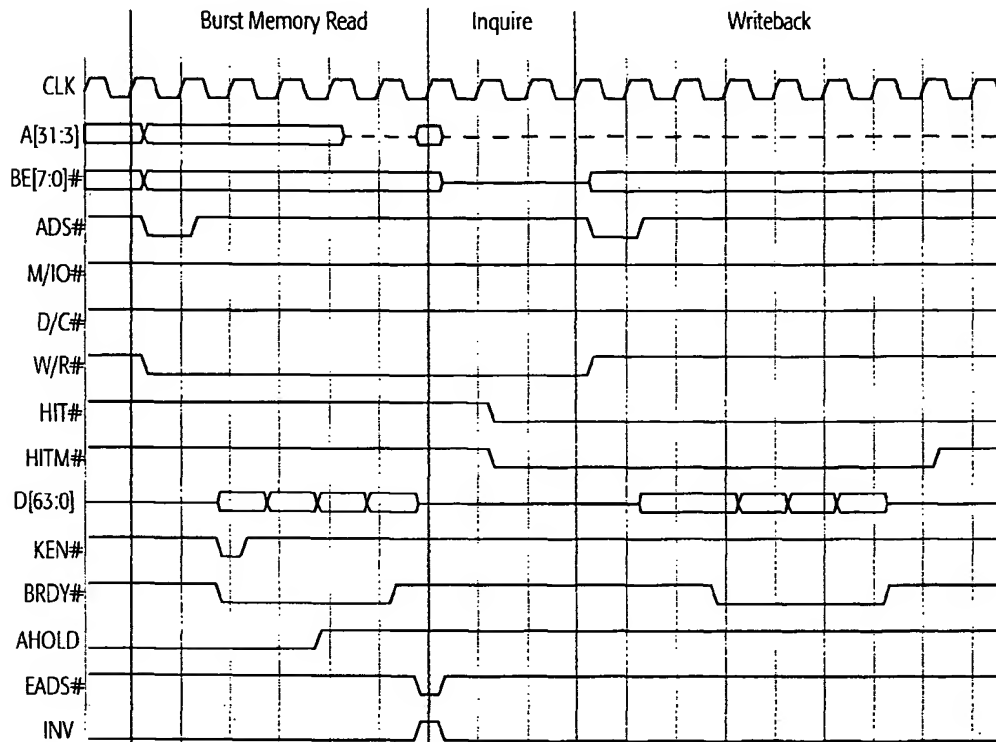
---

### **AHOLD-Initiated Inquire Hit to Modified Line**

Figure 65 on page 211 shows an AHOLD-initiated inquire cycle that hits a modified line. During the inquire cycle in this example, the processor asserts both HIT# and HITM# on the clock edge after the clock edge that it samples EADS# asserted. This condition indicates that the cache line exists in the processor's data cache in the modified state.

If the inquire cycle hits a modified line, the processor performs a writeback cycle immediately after the inquire cycle to update the modified cache line to shared memory (normally level-two cache or DRAM). In Figure 65, the system logic holds AHOLD asserted throughout the inquire cycle and the processor writeback cycle. In this case, the processor is not driving the address bus during the writeback cycle because AHOLD is sampled asserted. The system logic writes the data to memory by using its latched copy of the inquire cycle address. If the processor samples AHOLD negated before it performs the writeback cycle, it drives the writeback cycle by using the address (A[31:5]) that it latched during the inquire cycle.

If INV is sampled asserted during an inquire cycle, the processor transitions the line (if found) to the invalid state, regardless of its previous state (the cache invalidation operation is not visible on the bus). If INV is sampled negated during an inquire cycle, the processor transitions the line (if found) to the shared state. In either case, if the line is found in the modified state, the processor writes it back to memory before changing its state. Figure 65 shows that the processor samples INV asserted during the inquire cycle and invalidates the cache line after the inquire cycle.



**Figure 65. AHOLD-Initiated Inquire Hit to Modified Line**



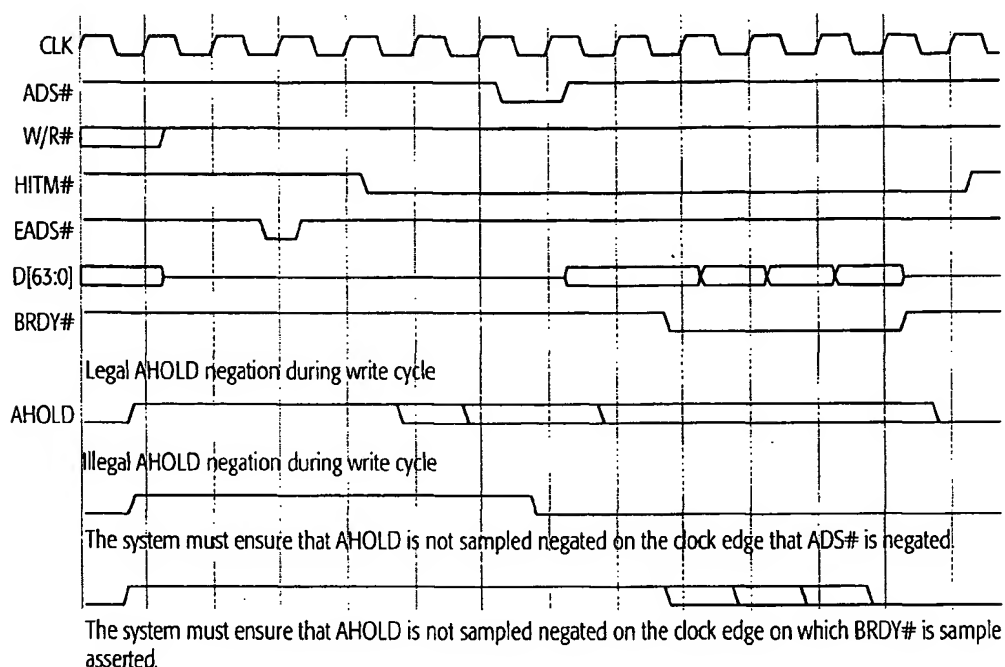
## **6** *Bus Cycles*

---

### **AHOLD Restriction**

When the system logic drives an AHOLD-initiated inquire cycle, it must assert AHOLD for at least two clocks before it asserts EADS#. This requirement guarantees the processor recognizes and responds to the inquire cycle properly. The processor's 32 address bus drivers turn on almost immediately after AHOLD is sampled negated. If the processor switches the data bus (D[63:0] and DP[7:0]) during a write cycle off the same clock edge that switches the address bus (A[31:3] and AP), the processor switches 102 drivers simultaneously, which can lead to ground-bounce spikes. Therefore, before negating AHOLD the following restrictions must be observed by the system logic:

- When the system logic negates AHOLD during a write cycle, it must ensure that AHOLD is not sampled negated on the clock edge on which BRDY# is sampled asserted (See Figure 66 on page 213).
- When the system logic negates AHOLD during a writeback cycle, it must ensure that AHOLD is not sampled negated on the clock edge on which ADS# is negated (See Figure 66).
- When a write cycle is pipelined into a read cycle, AHOLD must not be sampled negated on the clock edge after the clock edge on which the last BRDY# of the read cycle is sampled asserted to avoid the processor simultaneously driving the data bus (for the pending write cycle) and the address bus off this same clock edge.



**Figure 66. AHOLD Restriction**

## Bus Backoff (BOFF#)

**BOFF#** provides the fastest response among bus-hold inputs. Either the system logic or another bus master can assert **BOFF#** to gain control of the bus immediately. **BOFF#** is also used to resolve potential deadlock problems that arise as a result of inquire cycles. The processor samples **BOFF#** on every clock edge. If **BOFF#** is sampled asserted, the processor unconditionally aborts any cycles in progress and transitions to a bus hold state. (See “**BOFF#** (Backoff)” on page 148.) Figure 67 on page 214 shows a read cycle that is aborted when the processor samples **BOFF#** asserted even though **BRDY#** is sampled asserted on the same clock edge. The read cycle is restarted after **BOFF#** is sampled negated (**KEN#** must be in the same state during the restarted cycle as its state during the aborted cycle).

## 6 Bus Cycles

During a **BOFF#**-initiated inquire cycle that hits a shared or exclusive line, the processor samples **BOFF#** negated and restarts any bus cycle that was aborted when **BOFF#** was asserted. If a **BOFF#**-initiated inquire cycle hits a modified line, the processor performs a writeback cycle before it restarts the aborted cycle.

If the processor samples **BOFF#** asserted on the same clock edge that it asserts **ADS#**, **ADS#** is floated but the system logic may erroneously interpret **ADS#** as asserted. In this case, the system logic must properly interpret the state of **ADS#** when **BOFF#** is negated.

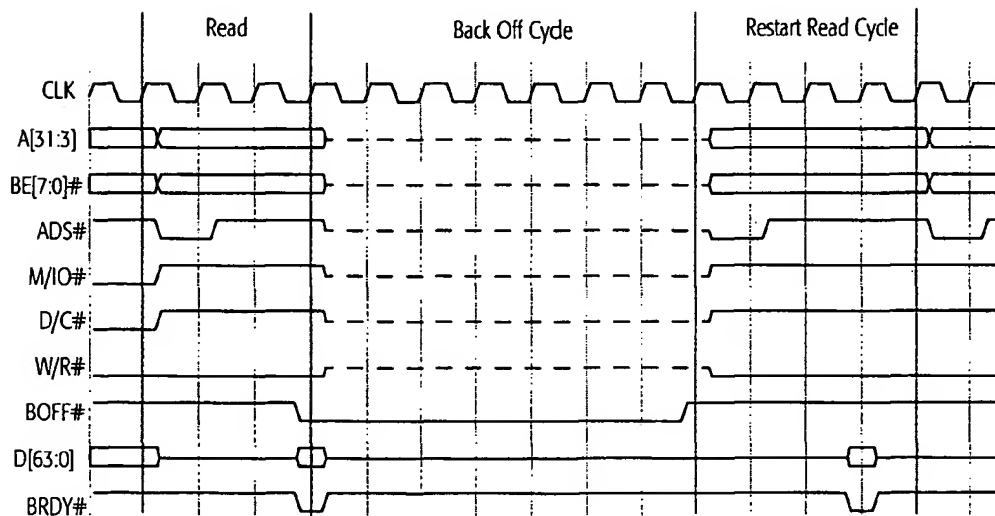


Figure 67. **BOFF#** Timing

## Locked Cycles

The processor asserts LOCK# during a sequence of bus cycles to ensure the cycles are completed without allowing other bus masters to intervene. Locked operations can consist of two to five cycles. LOCK# is asserted during the following operations:

- An interrupt acknowledge sequence
- Descriptor Table accesses
- Page Directory and Page Table accesses
- XCHG instruction
- An instruction with an allowable LOCK prefix

In order to ensure that locked operations appear on the bus and are visible to the entire system, any data operands addressed during a locked cycle that reside in the processor's cache are flushed and invalidated from the cache prior to the locked operation. If the cache line is in the modified state, it is written back and invalidated prior to the locked operation. Likewise, any data read during a locked operation is not cached. The processor negates LOCK# for at least one clock between consecutive sequences of locked operations to allow the system logic to arbitrate for the bus.

The processor asserts SCYC during misaligned locked transfers on the D[63:0] data bus. The processor generates additional bus cycles to complete the transfer of misaligned data.

## Basic Locked Operation

Figure 68 on page 216 shows a pair of read-write bus cycles. It represents a typical read-modify-write locked operation. The processor asserts LOCK# off the same clock edge that it asserts ADS# of the first bus cycle in the locked operation and holds it asserted until the last expected BRDY# of the last bus cycle in the locked operation is sampled asserted. (The processor negates LOCK# off the same clock edge.)

## 6 Bus Cycles

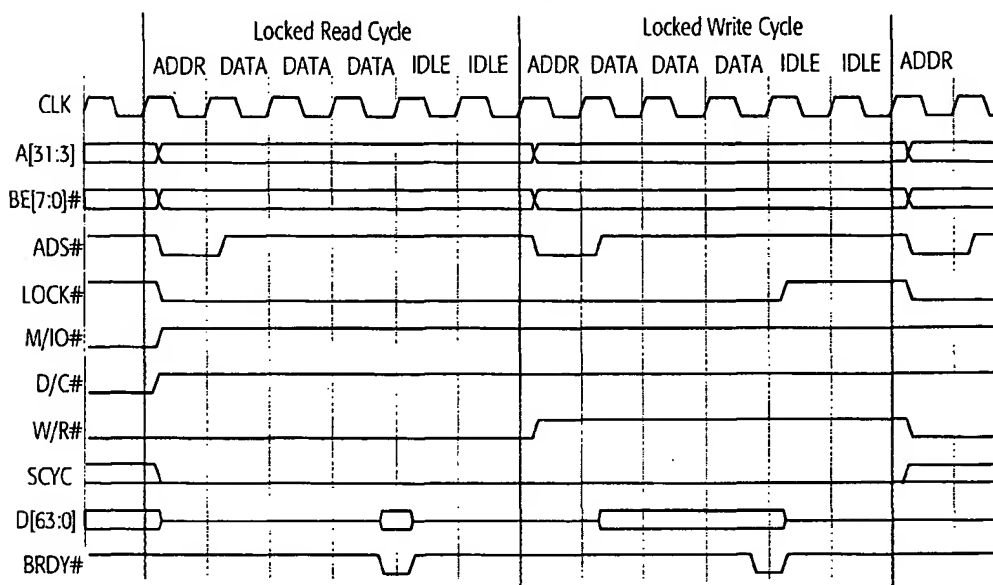


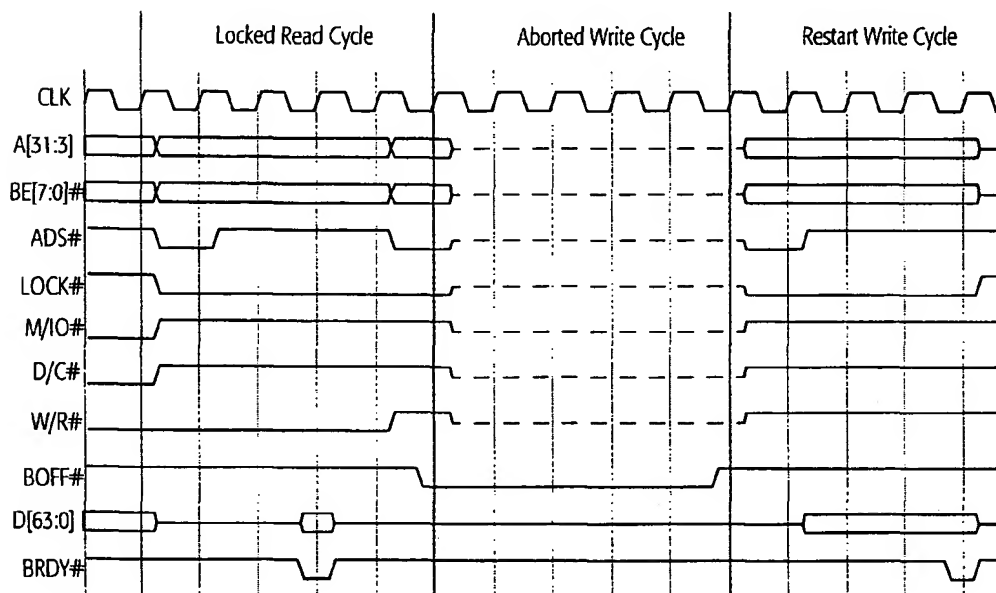
Figure 68. Basic Locked Operation

### Locked Operation with BOFF# Intervention

Figure 69 on page 217 shows BOFF# asserted within a locked read-write pair of bus cycles. In this example, the processor asserts LOCK# with ADS# to drive a locked memory read cycle followed by a locked memory write cycle. During the locked memory write cycle in this example, the processor samples BOFF# asserted. The processor immediately aborts the locked memory write cycle and floats all its bus-driving signals, including LOCK#. The system logic or another bus master can initiate an inquire cycle or drive a new bus cycle one clock edge after the clock edge on which BOFF# is sampled asserted. If the system logic drives a BOFF#-initiated inquire cycle and hits a modified line, the processor performs a writeback cycle before it restarts the locked cycle (the processor asserts LOCK# during the writeback cycle).

In Figure 69 on page 217, the processor immediately restarts the aborted locked write cycle by driving the bus off the clock edge on which BOFF# is sampled negated. The system logic

must ensure the processor results for interrupted and uninterrupted locked cycles are consistent. That is, the system logic must guarantee the memory accessed by the processor is not modified during the time another bus master controls the bus.



**Figure 69. Locked Operation with BOFF# Intervention**

## 6 *Bus Cycles*

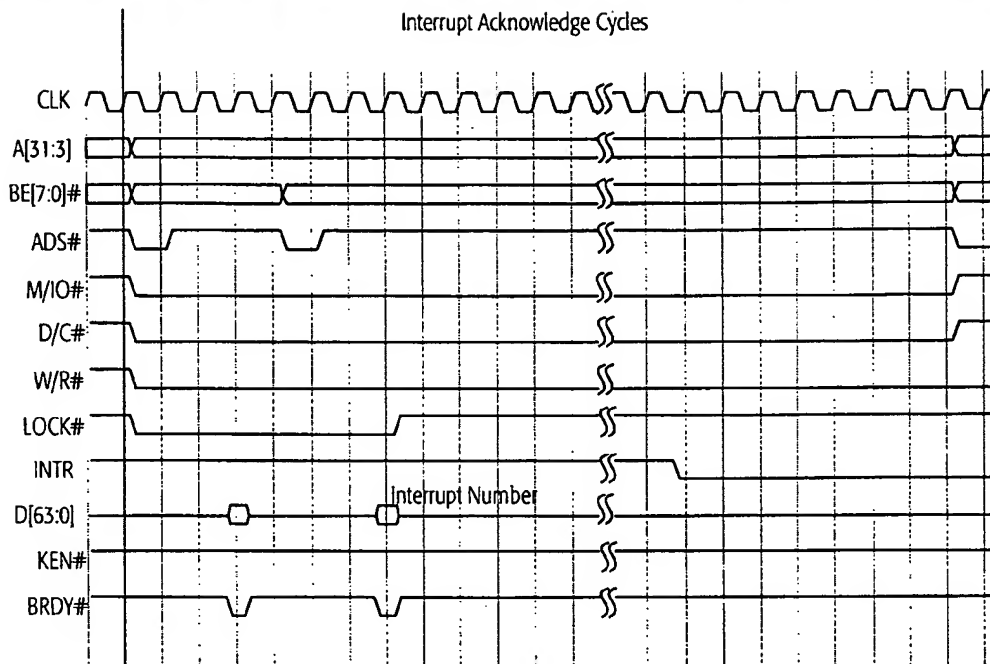
### Interrupt Acknowledge

In response to recognizing the system's maskable interrupt (INTR), the processor drives an interrupt acknowledge cycle at the next instruction boundary. During an interrupt acknowledge cycle, the processor drives a locked pair of read cycles as shown in Figure 70 on page 219. The first read cycle is not functional, and the second read cycle returns the interrupt number on D[7:0] (00h–FFh). Table 48 shows the state of the signals during an interrupt acknowledge cycle.

**Table 48. Interrupt Acknowledge Operation Definition**

Processor Outputs	First Bus Cycle	Second Bus Cycle
D/C#	Low	Low
M/IO#	Low	Low
W/R#	Low	Low
BE[7:0]#	EFh	FEh (low byte enabled)
A[31:3]	0000_0000h	0000_0000h
D[63:0]	(ignored)	Interrupt number expected from interrupt controller on D[7:0]

The system logic can drive INTR either synchronously or asynchronously. If it is asserted asynchronously, it must be asserted for a minimum pulse width of two clocks. To ensure it is recognized, INTR must remain asserted until an interrupt acknowledge sequence is complete.



**Figure 70. Interrupt Acknowledge Operation**



# 6 Bus Cycles

## Special Bus Cycles

The AMD-K6 3D processor drives special bus cycles that include stop grant, flush acknowledge, cache writeback invalidation, halt, cache invalidation, and shutdown cycles. During all special cycles, D/C# = 0, M/IO# = 0, and W/R# = 1. BE[7:0]# and A[31:3] are driven to differentiate among the special cycles, as shown in Table 49. The system logic must return BRDY# in response to all processor special cycles.

**Table 49. Encodings For Special Bus Cycles**

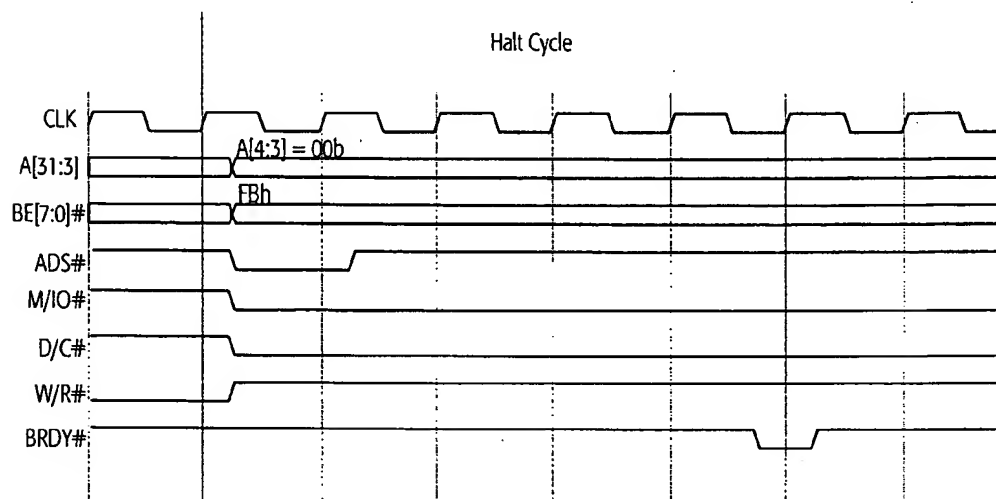
BE[7:0]#	A[4:3]*	Special Bus Cycle	Cause
FBh	10b	Stop Grant	STPCLK# sampled asserted
EFh	00b	Flush Acknowledge	FLUSH# sampled asserted
F7h	00b	Writeback	WBINVD instruction
FBh	00b	Halt	HLT instruction
FDh	00b	Flush	INVD, WBINVD instruction
FEh	00b	Shutdown	Triple fault
<b>Note:</b> * A[31:5] = 0			

### Basic Special Bus Cycle

Figure 71 on page 221 shows a basic special bus cycle. The processor drives D/C# = 0, M/IO# = 0, and W/R# = 1 off the same clock edge that it asserts ADS#. In this example, BE[7:0]# = FBh and A[31:3] = 0000\_0000h, which indicates that the special cycle is a halt special cycle (See Table 49). A halt special cycle is generated after the processor executes the HLT instruction.

If the processor samples FLUSH# asserted, it writes back any data cache lines that are in the modified state and invalidates all lines in the instruction and data cache. The processor then drives a flush acknowledge special cycle.

If the processor executes a WBINVD instruction, it drives a writeback special cycle after the processor completes invalidating and writing back the cache lines.



**Figure 71. Basic Special Bus Cycle (Halt Cycle)**

## 6 Bus Cycles

### Shutdown Cycle

In Figure 72, a shutdown (triple fault) occurs in the first half of the waveform, and a shutdown special cycle follows in the second half. The processor enters shutdown when an interrupt or exception occurs during the handling of a double fault (INT 8), which amounts to a triple fault. When the processor encounters a triple fault, it stops its activity on the bus and generates the shutdown special bus cycle (BE[7:0]# = FEh).

The system logic must assert NMI, INIT, RESET, or SMI# to get the processor out of the shutdown state.

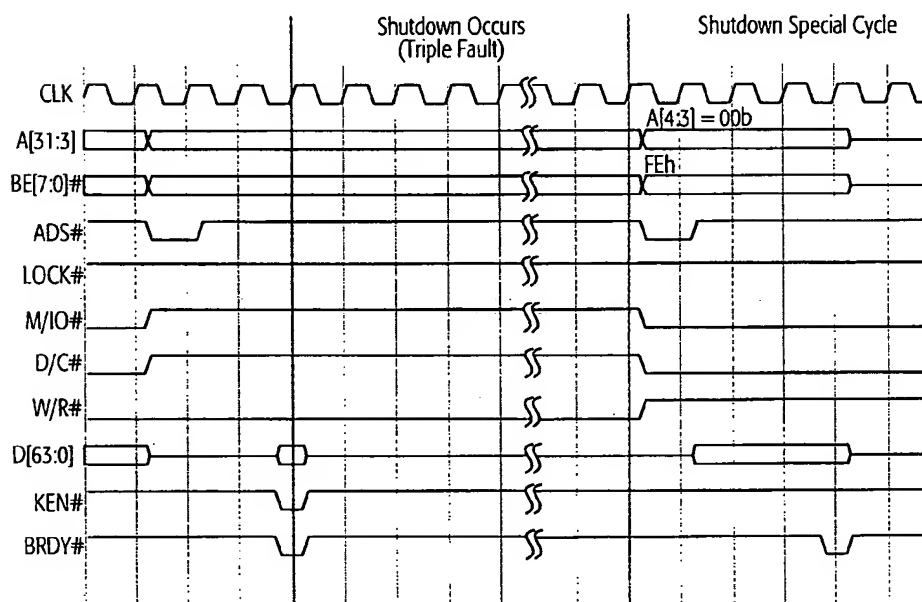


Figure 72. Shutdown Cycle

## Stop Grant and Stop Clock States

Figure 73 on page 224 and Figure 74 on page 225 show the processor transition from normal execution to the Stop Grant state, then to the Stop Clock state, back to the Stop Grant state, and finally back to normal execution. The series of transitions begins when the processor samples STPCLK# asserted. On recognizing a STPCLK# interrupt at the next instruction retirement boundary, the processor performs the following actions, in the order shown:

1. Its instruction pipelines are flushed
2. All pending and in-progress bus cycles are completed
3. The STPCLK# assertion is acknowledged by executing a Stop Grant special bus cycle
4. Its internal clock is stopped after BRDY# of the Stop Grant special bus cycle is sampled asserted and after EWBE# is sampled asserted
5. The Stop Clock state is entered if the system logic stops the bus clock CLK (optional)

STPCLK# is sampled as a level-sensitive input on every clock edge but is not recognized until the next instruction boundary. The system logic drives the signal either synchronously or asynchronously. If it is asserted asynchronously, it must be asserted for a minimum pulse width of two clocks. STPCLK# must remain asserted until recognized, which is indicated by the completion of the Stop Grant special cycle.

## 6 Bus Cycles

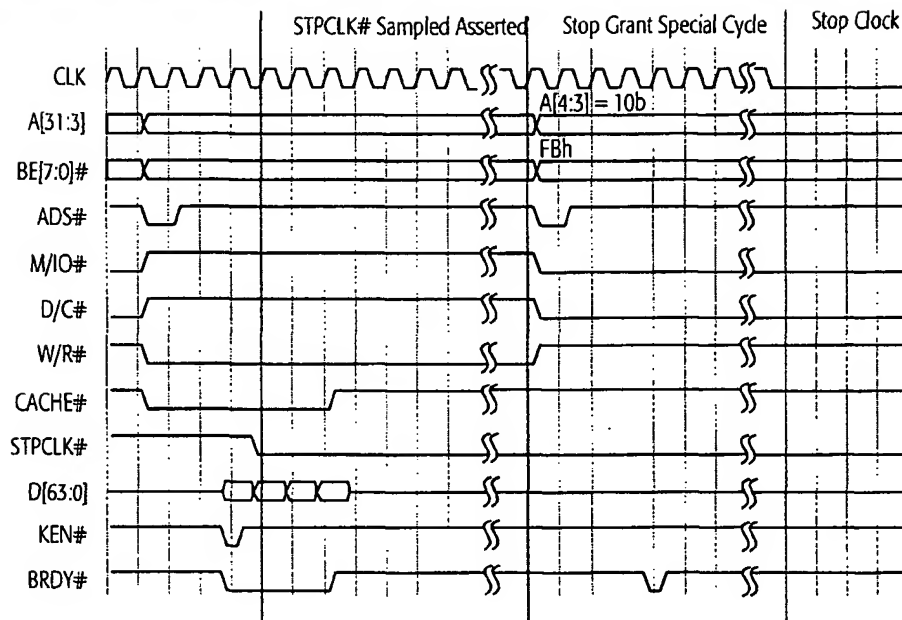


Figure 73. Stop Grant and Stop Clock Modes, Part 1

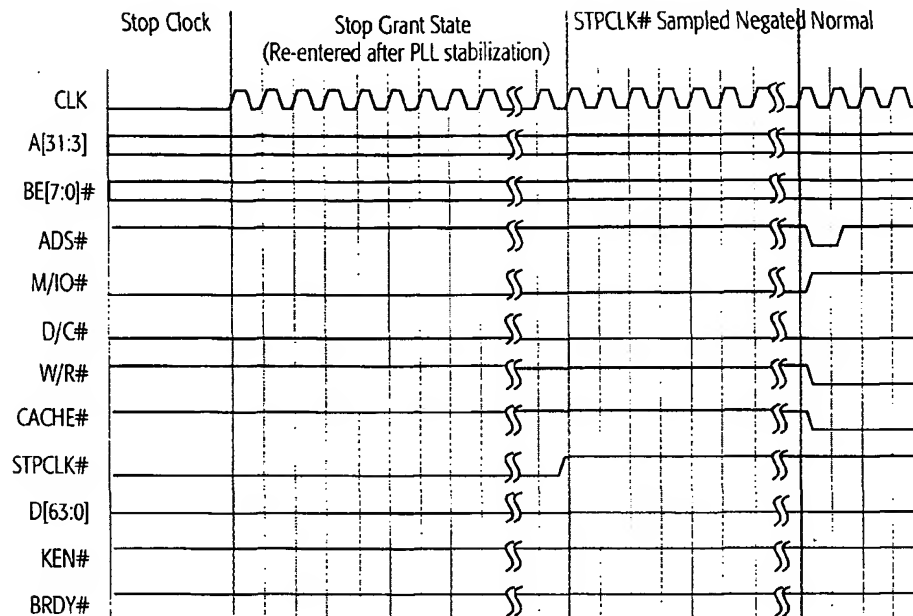


Figure 74. Stop Grant and Stop Clock Modes, Part 2

## INIT-Initiated Transition from Protected Mode to Real Mode

INIT is typically asserted in response to a BIOS interrupt that writes to an I/O port. This interrupt is often in response to a Ctrl-Alt-Del keyboard input. The BIOS writes to a port (similar to port 64h in the keyboard controller) that asserts INIT. INIT is also used to support 80286 software that must return to Real mode after accessing extended memory in Protected mode.

The assertion of INIT causes the processor to empty its pipelines, initialize most of its internal state, and branch to address FFFF\_FFF0h—the same instruction execution starting point used after RESET. Unlike RESET, the processor preserves the contents of its caches, the floating-point state, the MMX state, model-specific registers (MSRs), the CD and NW bits of the CR0 register, the time stamp counter, and other specific internal resources.

## 6 Bus Cycles

Figure 75 shows an example in which the operating system writes to an I/O port, causing the system logic to assert INIT. The sampling of INIT asserted starts an extended microcode sequence that terminates with a code fetch from FFFF\_FFF0h, the reset location. INIT is sampled on every clock edge but is not recognized until the next instruction boundary. During an I/O write cycle, it must be sampled asserted a minimum of three clock edges before BRDY# is sampled asserted if it is to be recognized on the boundary between the I/O write instruction and the following instruction. If INIT is asserted synchronously, it can be asserted for a minimum of one clock. If it is asserted asynchronously, it must have been negated for a minimum of two clocks, followed by an assertion of a minimum of two clocks.

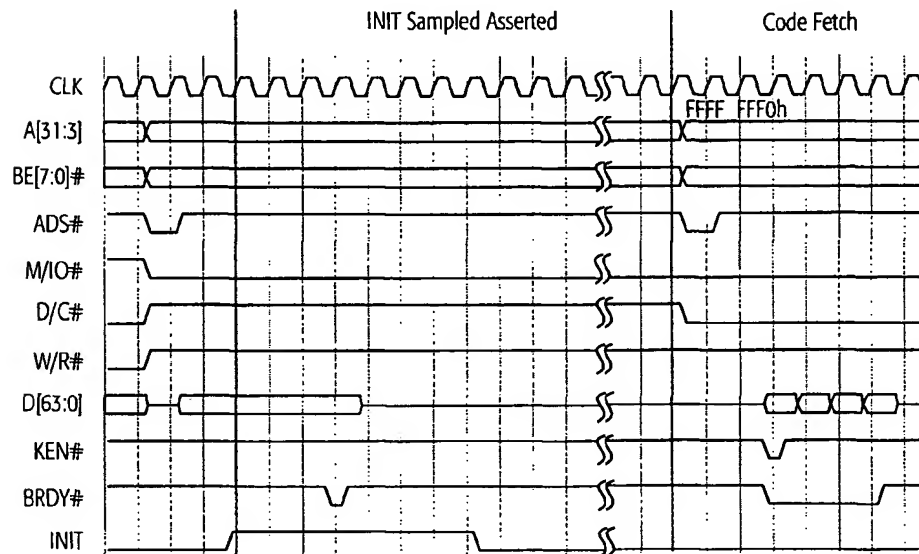


Figure 75. INIT-Initiated Transition from Protected Mode to Real Mode

# 7

## Power-on Configuration and Initialization

On power-on the system logic must reset the AMD-K6 3D processor by asserting the RESET signal. When the processor samples RESET asserted, it immediately flushes and initializes all internal resources and its internal state, including its pipelines and caches, the floating-point state, the MMX and 3D states, and all registers. Then the processor jumps to address FFFF\_FFF0h to start instruction execution.

### Signals Sampled During the Falling Transition of RESET

#### **FLUSH#**

FLUSH# is sampled on the falling transition of RESET to determine if the processor begins normal instruction execution or enters Tri-State Test mode. If FLUSH# is High during the falling transition of RESET, the processor unconditionally runs its Built-In Self Test (BIST), performs the normal reset functions, then jumps to address FFFF\_FFF0h to start instruction execution. (See “Built-In Self-Test (BIST)” on page 270 for more details.) If FLUSH# is Low during the falling transition of RESET, the processor enters Tri-State Test mode. (See “Tri-State Test Mode” on page 270 and “FLUSH# (Cache Flush)” on page 159 for more details.)



# **7** *Power-on Configuration and Initialization*

---

## **BF[2:0]**

The internal operating frequency of the processor is determined by the state of the bus frequency signals BF[2:0] when they are sampled during the falling transition of RESET. The frequency of the CLK input signal is multiplied internally by a ratio defined by BF[2:0]. (“BF[2:0] (Bus Frequency)” on page 147 for the processor-clock to bus-clock ratios.)

## **BRDYC#**

BRDYC# is sampled on the falling transition of RESET to configure the drive strength of A[20:3], ADS#, HITM#, and W/R#. If BRDYC# is Low during the fall of RESET, these outputs are configured using higher drive strengths than the standard strength. If BRDYC# is High during the fall of RESET, the standard strength is selected. (See “BRDYC# (Burst Ready Copy)” on page 150 for more details.)

## **RESET Requirements**

---

During the initial power-on reset of the processor, RESET must remain asserted for a minimum of 1.0 ms after CLK and V<sub>CC</sub> reach specification. (See “CLK Switching Characteristics” on page 314 for clock specifications. See Chapter 14, “Electrical Data” on page 303 for V<sub>CC</sub> specifications.)

During a warm reset while CLK and V<sub>CC</sub> are within specification, RESET must remain asserted for a minimum of 15 clocks prior to its negation.

## State of Processor After RESET

### Output Signals

Table 50 shows the state of all processor outputs and bidirectional signals immediately after RESET is sampled asserted.

**Table 50. Output Signal State After RESET**

Signal	State	Signal	State
A[31:3], AP	Floating	LOCK#	High
ADS#, ADSC#	High	M/IO#	Low
APCHK#	High	PCD	Low
BE[7:0]#	Floating	PCHK#	High
BREQ	Low	PWT	Low
CACHE#	High	SCYC	Low
D/C#	Low	SMIACK#	High
D[63:0], DP[7:0]	Floating	TDO	Floating
FERR#	High	VCC2DET	Low
HIT#	High	VCC2H/L#	Low
HITM#	High	W/R#	Low
HLDA	Low	—	—

### Registers

Table 51 on page 230 shows the state of all architecture registers and model-specific registers (MSRs) after the processor has completed its initialization due to the recognition of the assertion of RESET.

# 7 Power-on Configuration and Initialization

**Table 51. Register State After RESET**

Register	State (hex)	Notes
GDTR	base:0000_0000h limit:0FFFFh	
IDTR	base:0000_0000h limit:0FFFFh	
TR	0000h	
LDTR	0000h	
EIP	FFFF_FFF0h	
EFLAGS	0000_0002h	
EAX	0000_0000h	1
EBX	0000_0000h	
ECX	0000_0000h	
EDX	0000_058Xh	2
ESI	0000_0000h	
EDI	0000_0000h	
EBP	0000_0000h	
ESP	0000_0000h	
CS	F000h	
SS	0000h	
DS	0000h	
ES	0000h	
FS	0000h	
GS	0000h	
FPU Stack R7–R0	0000_0000_0000_0000_0000h	3
FPU Control Word	0040h	3
FPU Status Word	0000h	3
FPU Tag Word	5555h	3
FPU Instruction Pointer	0000_0000_0000h	3
FPU Data Pointer	0000_0000_0000h	3
FPU Opcode Register	000_0000_0000b	3
CR0	6000_0010h	4
CR2	0000_0000h	

**Notes:**

1. The contents of EAX indicate if BIST was successful. If EAX = 0000\_0000h, BIST was successful. If EAX is non-zero, BIST failed.
2. EDX contains the AMD-K6 processor signature, where X indicates the processor Stepping ID.
3. The contents of these registers are preserved following the recognition of INIT.
4. The CD and NW bits of CR0 are preserved following the recognition of INIT.

**Table 51. Register State After RESET (continued)**

Register	State (hex)	Notes
CR3	0000_0000h	
CR4	0000_0000h	
DR7	0000_0400h	
DR6	FFFF_0FF0h	
DR3	0000_0000h	
DR2	0000_0000h	
DR1	0000_0000h	
DR0	0000_0000h	
MCAR	0000_0000_0000_0000h	3
MCTR	0000_0000_0000_0000h	3
TR12	0000_0000_0000_0000h	3
TSC	0000_0000_0000_0000h	3
EFER	0000_0000_0000_0000h	3
STAR	0000_0000_0000_0000h	3
WHCR	0000_0000_0000_0000h	3
<b>Notes:</b> <ol style="list-style-type: none"> <li>1. The contents of EAX indicate if BIST was successful. If EAX = 0000_0000h, BIST was successful. If EAX is non-zero, BIST failed.</li> <li>2. EDX contains the AMD-K6 processor signature, where X indicates the processor Stepping ID.</li> <li>3. The contents of these registers are preserved following the recognition of INIT.</li> <li>4. The CD and NW bits of CR0 are preserved following the recognition of INIT.</li> </ol>		

---

## **7** *Power-on Configuration and Initialization*

---

### **State of Processor After INIT**

---

The recognition of the assertion of INIT causes the processor to empty its pipelines, to initialize most of its internal state, and to branch to address FFFF\_FFF0h—the same instruction execution starting point used after RESET. Unlike RESET, the processor preserves the contents of its caches, the floating-point state, the MMX and 3D states, MSRs, and the CD and NW bits of the CR0 register.

The edge-sensitive interrupts FLUSH# and SMI# are sampled and preserved during the INIT process and are handled accordingly after the initialization is complete. However, the processor resets any pending NMI interrupt upon sampling INIT asserted.

INIT can be used as an accelerator for 80286 code that requires a reset to exit from Protected mode back to Real mode.

## Cache Organization

The following sections describe the basic architecture and resources of the AMD-K6 3D processor internal caches.

The performance of the processor is enhanced by a writeback level-one (L1) cache. The cache is organized as a separate 32-Kbyte instruction cache and a 32-Kbyte data cache, each with two-way set associativity (See Figure 76 on page 234). The cache line size is 32 bytes, and lines are prefetched from main memory using an efficient, pipelined burst transaction. As the instruction cache is filled, each instruction byte is analyzed for instruction boundaries using predecode logic. Predecoding annotates each instruction byte with information that later enables the decoders to efficiently decode multiple instructions simultaneously. Translation lookaside buffers (TLB) are also used to translate linear addresses to physical addresses. The instruction cache is associated with a 64-entry TLB while the data cache is associated with a 128-entry TLB.

# 8 Cache Organization

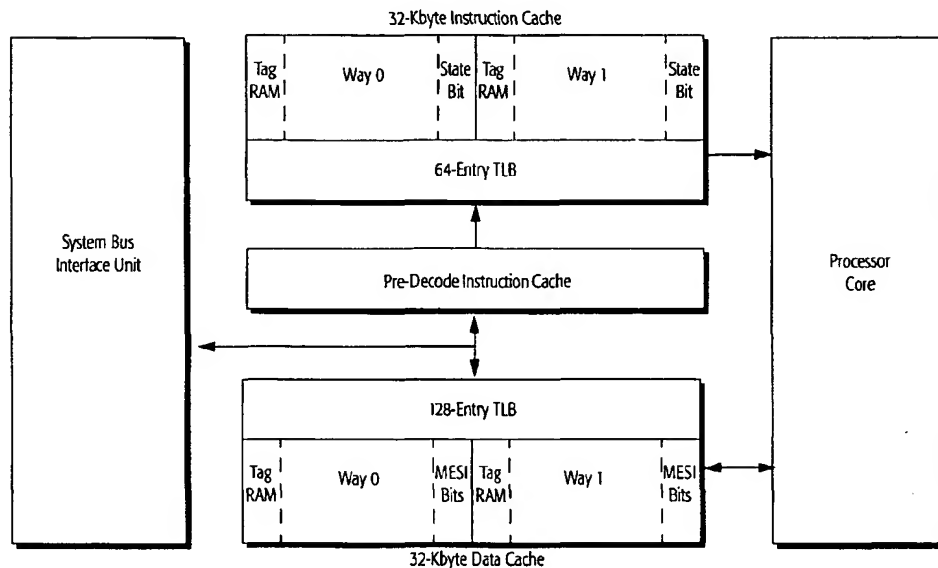


Figure 76. Cache Organization

The processor cache design takes advantage of a sectored organization (See Figure 77). Each sector consists of 64 bytes configured as two 32-byte cache lines. The two cache lines of a sector share a common tag but have separate MESI (modified, exclusive, shared, invalid) bits that track the state of each cache line.

## Instruction Cache Line

Tag	Cache Line 1	Byte 31	Predecode Bits	Byte 30	Predecode Bits	.....	.....	Byte 0	Predecode Bits	1 MESI Bit
Address	Cache Line 2	Byte 31	Predecode Bits	Byte 30	Predecode Bits	.....	.....	Byte 0	Predecode Bits	1 MESI Bit

## Data Cache Line

Tag	Cache Line 1	Byte 31	Byte 30	.....	.....	Byte 0	2 MESI Bits
Address	Cache Line 2	Byte 31	Byte 30	.....	.....	Byte 0	2 MESI Bits

**Note:** Instruction-cache lines have only two coherency states (valid or invalid) rather than the four MESI coherency states of data-cache lines. Only two states are needed for the instruction cache because these lines are read-only.

Figure 77. Cache Sector Organization

## MESI States in the Data Cache

---

The state of each line in the caches is tracked by the MESI bits. The coherency of these states or MESI bits is maintained by internal processor snoops and external inquiries by the system logic. The following four states are defined for the data cache:

- *Modified*—This line has been modified and is different from main memory.
- *Exclusive*—This line is not modified and is the same as main memory. If this line is written to, it becomes Modified.
- *Shared*—If a cache line is in the shared state it means that the same line can exist in more than one cache system.
- *Invalid*—The information in this line is not valid.

## Predecode Bits

---

Decoding x86 instructions is particularly difficult because the instructions vary in length, ranging from 1 to 15 bytes long. Predecode logic supplies the predecode bits associated with each instruction byte. The predecode bits indicate the number of bytes to the start of the next x86 instruction. The predecode bits are passed with the instruction bytes to the decoders where they assist with parallel x86 instruction decoding. The predecode bits use memory separate from the 32-Kbyte instruction cache. The predecode bits are stored in an extended instruction cache alongside each x86 instruction byte as shown in Figure 77 on page 234.

## Cache Operation

---

The operating modes for the caches are configured by software using the not writethrough (NW) and cache disable (CD) bits of control register 0 (CR0 bits 29 and 30 respectively). These bits are used in all operating modes.

When the CD and NW bits are both set to 0, the cache is fully enabled. This is the standard operating mode for the cache. If a



## 8 Cache Organization

---

read miss occurs when the processor reads from the cache, a line fill takes place. Write hits to the cache are updated, while write misses and writes to shared lines cause external memory updates.

*Note: A write allocate operation can modify the behavior of write misses to the cache. See "Write Allocate" on page 240*

When CD is set to 0 and NW is set to 1, an invalid mode of operation exists that causes a general protection fault to occur.

When CD is set to 1 (disabled) and NW is set to 0, the cache fill mechanism is disabled but the contents of the cache are still valid. The processor reads from the cache and, if a read miss occurs, no line fills take place. Write hits to the cache are updated, while write misses and writes to shared lines cause external memory updates.

When the CD and NW bits are both set to 1, the cache is fully disabled. Even though the cache is disabled, the contents are not necessarily invalid. The processor reads from the cache and, if a read miss occurs, no line fills take place. If a write hit occurs, the cache is updated but an external memory update does not occur. If a data line is in the exclusive state during a write hit, the MESI bits are changed to the modified state. Write misses access memory directly.

The operating system can control the cacheability of a page. The paging mechanism is controlled by CR3, the Page Directory Entry (PDE), and the Page Table Entry (PTE). Within CR3, PDE, and PTE are Page Cache Disable (PCD) and Page Writethrough (PWT) bits. The values of the PCD and PWT bits used in Table 52 through Table 54 on page 237 are taken from either the PTE or PDE. For more information see the descriptions of PCD and PWT on pages 171 and 173, respectively.

Table 52 through Table 54 describe the logic that determines the cacheability of a cycle and how that cacheability is affected by the PCD bits, the PWT bits, the PG bit of CR0, the CD bit of CR0, writeback cycles, the Cache Inhibit (CI) bit of Test Register 12 (TR12), and unlocked memory reads.

Table 52 describes how the PWT signal is driven based on the values of the PWT bits and the PG bit of CR0.

**Table 52. PWT Signal Generation**

PWT Bit*	PG Bit of CR0	PWT Signal
1	1	High
0	1	Low
1	0	Low
0	0	Low
<b>Note:</b> * PWT is taken from PTE or PDE		

Table 53 describes how the PCD signal is driven based on the values of the CD bit of CR0, the PCD bits, and the PG bit of CR0.

**Table 53. PCD Signal Generation**

CD Bit of CR0	PCD Bit*	PG Bit of CR0	PCD Signal
1	X	X	High
0	1	1	High
0	0	1	Low
0	1	0	Low
0	0	0	Low
<b>Note:</b> * PCD is taken from PTE or PDE			

Table 54 describes how the CACHE# signal is driven based on writeback cycles, the CI bit of TR12, unlocked memory reads, and the PCD signal.

**Table 54. CACHE# Signal Generation**

Writeback Cycle	CI Bit of TR12	Unlocked Memory Reads	PCD Signal	CACHE#
1	X	X	X	Low
0	1	1	High	High
0	0	1	High	High
0	1	0	High	High
0	0	0	High	High
0	1	1	Low	High

## 8 Cache Organization

Table 54. CACHE# Signal Generation (continued)

Writeback Cycle	CI Bit of TR12	Unlocked Memory Reads	PCD Signal	CACHE#
0	0	1	Low	Low
0	1	0	Low	High
0	0	0	Low	High

### Cache-Related Signals

Complete descriptions of the signals that control cacheability and cache coherency are given on the following pages:

- CACHE#—page 152
- EADS#—page 156
- FLUSH#—page 159
- HIT#—page 160
- HITM#—page 160
- INV—page 165
- KEN#—page 166
- PCD—page 171
- PWT—page 173
- WB/WT#—page 183

### Cache Disabling

To completely disable all cache accesses, the CD and NW bits must be set to 1 and the cache must be completely flushed.

There are two different methods for flushing the cache. The first method relies on the system logic and the second relies on software.

For the system logic to flush the cache, the processor must sample FLUSH# asserted. In this method, the processor writes back any data cache lines that are in the modified state, invalidates all lines in the instruction and data caches, and then executes a flush acknowledge special cycle (See Table 44 on page 186).

Software can use two different instructions to flush the cache. Both the WBINVD and INVD instructions cause all cache lines to be marked invalid. The WBINVD instruction causes all modified lines to first be written back to memory. The INVD instruction invalidates all cache lines without writing modified lines back to memory.

Any area of system memory can be cached. However, the processor prevents caching of locked operations and TLB reads, the operating system can prevent caching of certain pages by setting the PCD and PWT bits in the PDE or PTE, and system logic can prevent caching of certain bus cycles by negating the KEN# input signal with the first BRDY# or NA# of a cycle.

## **Cache-Line Fills**

---

When the processor needs to read memory, the processor drives a read cycle onto the bus. If the cycle is cacheable the processor asserts CACHE#. The system logic also has control of the cacheability of bus cycles. If it determines the address is cacheable, system logic asserts the KEN# signal and the appropriate value of WB/WT#.

One of two events takes place next. If the cycle is not cacheable, a non-pipelined, single-transfer read takes place. The processor waits for the system logic to return the data and assert a single BRDY# (See Figure 54 on page 193). If the cycle is cacheable, the processor executes a 32-byte burst read cycle. The processor expects a total of four BRDY# signals for a burst read cycle to take place (See Figure 56 on page 197).

Instruction-cache line fills initiate 32-byte transfers from memory (one burst cycle) on the bus. Data-cache line fills also initiate 32-byte transfers on the bus. If the data-cache line being filled replaces a modified line, the prior contents of the line are copied to a 32-byte writeback (copyback) buffer in the bus interface unit while the new line is being read.

## **8** *Cache Organization*

---

### **Cache-Line Replacements**

---

As programs execute and task switches occur, some cache lines eventually require replacement.

Instruction cache lines are replaced using a Least Recently Used (LRU) algorithm. If line replacement is required, lines are replaced when read cache misses occur.

The data cache uses a slightly different approach to line replacement. If a miss occurs, and a replacement is required, lines are replaced by using a Least Recently Allocated (LRA) algorithm.

Two forms of cache misses and associated cache fills can take place—a sector replacement and a cache line replacement. In the case of a sector replacement, the miss is due to a tag mismatch, in which case the required cache line is filled from external memory, and the cache line within the sector that was not required is marked as invalid. In the case of a cache line replacement, the address matches the tag, but the requested cache line is marked as invalid. The required cache line is filled from external memory, and the cache line within the sector that is not required remains in the same cache state.

### **Write Allocate**

---

Write allocate, if enabled, occurs when the processor has a pending memory write cycle to a cacheable line and the line does not currently reside in the L1 data cache. In this case, the processor performs a burst read cycle to fetch the data-cache line addressed by the pending write cycle. The data associated with the pending write cycle is merged with the recently-allocated data-cache line and stored in the processor's L1 data cache. The final MESI state of the cache line depends on the state of the WB/WT# and PWT signals during the burst read cycle and the subsequent cache write hit (See Table 55 on page 246 to determine the cache-line states and the access types following a cache read miss and cache write hit).

During write allocates, a 32-byte burst read cycle is executed in place of a non-burst write cycle. While the burst read cycle generally takes longer to execute than the write cycle, performance gains are realized on subsequent write cycle hits to the write-allocated cache line. Due to the nature of software, memory accesses tend to occur in proximity of each other (principle of locality). The likelihood of additional write hits to the write-allocated cache line is high.

The following is a description of three mechanisms by which the AMD-K6 3D processor performs write allocations. A write allocate is performed when any one or more of these mechanisms indicates that a pending write is to a cacheable area of memory.

## **Write to a Cacheable Page**

Every time the processor performs a cache line fill, the address of the page in which the cache line resides is saved in the Cacheability Control Register (CCR). The page address of subsequent write cycles is compared with the page address stored in the CCR. If the two addresses are equal, then the processor performs a write allocate because the page has already been determined to be cacheable.

When the processor performs a cache line fill from a different page than the address saved in the CCR, the CCR is updated with the new page address.

## **Write to a Sector**

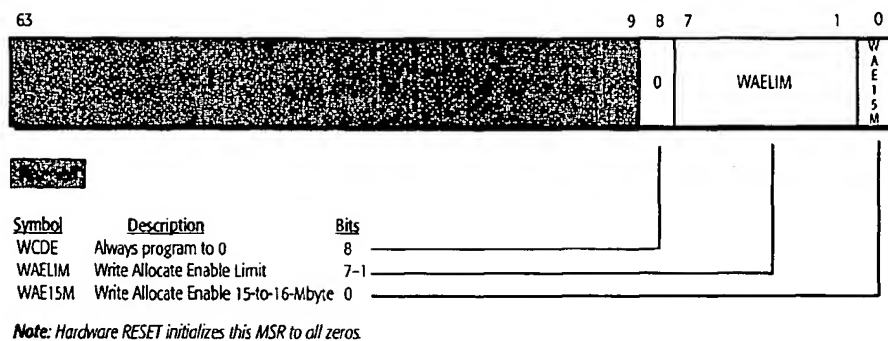
If the address of a pending write cycle matches the tag address of a valid cache sector, but the addressed cache line within the sector is marked invalid (a sector hit but a cache line miss), then the processor performs a write allocate. The pending write cycle is determined to be cacheable because the sector hit indicates the presence of at least one valid cache line in the sector. The two cache lines within a sector are guaranteed by design to be within the same page.

## 8 Cache Organization

### Write Allocate Limit

The Write Handling Control Register (WHCR) is a MSR that contains three fields—the WCDE bit, the Write Allocate Enable Limit (WAE LIM) field, and the Write Allocate Enable 15-to-16-Mbyte (WAE15M) bit (See Figure 78).

For proper functionality, always program the WCDE bit to 0.



**Figure 78. Write Handling Control Register (WHCR)**

The WAE LIM field is 7 bits wide. This field, multiplied by 4 Mbytes, defines an upper memory limit. Any pending write cycle that addresses memory below this limit causes the processor to perform a write allocate. Write allocate is disabled for memory accesses at and above this limit unless the processor determines a pending write cycle is cacheable by means of one of the other write allocate mechanisms—Write to a Cacheable Page and Write to a Sector. The maximum value of this memory limit is  $((2^7 - 1) \cdot 4 \text{ Mbytes}) = 508 \text{ Mbytes}$ . When all the bits in this field are set to 0, all memory is above this limit and this mechanism for allowing write allocate is effectively disabled.

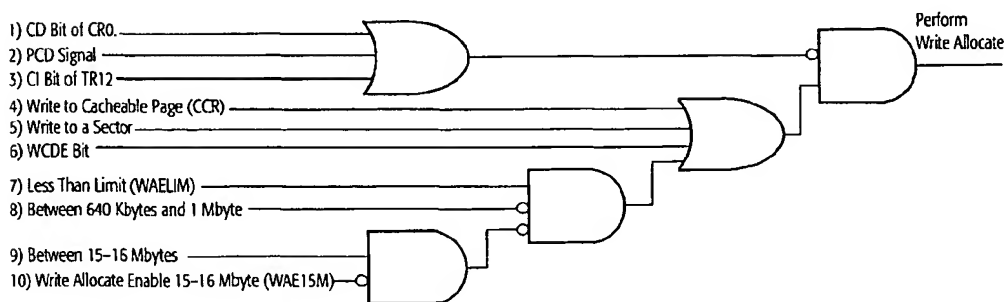
The Write Allocate Enable 15-to-16-Mbyte (WAE15M) bit is used to enable write allocations for the memory write cycles that address the 1 Mbyte of memory between 15 Mbytes and 16 Mbytes. This bit must be set to 1 to allow write allocate in this memory area. This bit is provided to account for a small number of uncommon memory-mapped I/O adapters that use this particular memory address space. If the system contains one of

these peripherals, the bit should be set to 0. The WAE15M bit is ignored if the value in the WAELIM field is set to less than 16 Mbytes.

By definition a write allocate is never performed in the memory area between 640 Kbytes and 1 Mbyte unless the processor determines a pending write cycle is cacheable by means of one of the other write allocate mechanisms—Write to a Cacheable Page and Write to a Sector. It is not considered safe to perform write allocations between 640 Kbytes and 1 Mbyte (000A\_0000h to 000F\_FFFFh) because it is considered a noncacheable region of memory.

Figure 79 shows the logic flow for all the mechanisms involved with write allocate for memory bus cycles. The left side of the diagram (the text) describes the conditions that need to be true in order for the value of that line to be a 1. Items 1 to 3 of the diagram are related to general cache operation and items 4 to 11 are related to the write allocate mechanisms.

For more information about write allocate, see the *Implementation of Write Allocate in the K86™ Processors Application Note*, document number 21326.



**Figure 79. Write Allocate Logic Mechanisms and Conditions**



## 8 Cache Organization

### Descriptions of the Logic Mechanisms and Conditions

1. *CD Bit of CR0*—When the cache disable (CD) bit within control register 0 (CR0) is set to 1, the cache fill mechanism for both reads and writes is disabled, therefore write allocate does not occur.
2. *PCD Signal*—When the PCD (page cache disable) signal is driven High, caching for that page is disabled even if KEN# is sampled asserted, therefore write allocate does not occur.
3. *CI Bit of TR12*—When the cache inhibit bit of Test Register 12 is set to 1, the L1 caches are disabled, therefore write allocate does not occur.
4. *Write to a Cacheable Page (CCR)*—A write allocate is performed if the processor knows that a page is cacheable. The CCR is used to store the page address of the last cache fill for a read miss. See “Write to a Cacheable Page” on page 241 for a detailed description of this condition.
5. *Write to a Sector*—A write allocate is performed if the address of a pending write cycle matches the tag address of a valid cache sector but the addressed cache line within the sector is invalid. See “Write to a Sector” on page 241 for a detailed description of this condition.
6. *WCDE Bit*—For proper functionality, always program bit 8 of WHCR to 0.
7. *Less Than Limit (WAE LIM)*—The write allocate limit mechanism determines if the memory area being addressed is less than the limit set in the WAE LIM field of WHCR. If the address is less than the limit, write allocate for that memory address is performed as long as conditions 9 and 10 do not prevent write allocate.
8. *Between 640 Kbytes and 1 Mbyte*—Write allocate is not performed in the memory area between 640 Kbytes and 1 Mbyte. It is not considered safe to perform write allocations between 640 Kbytes and 1 Mbyte (000A\_0000h to 000F\_FFFFh) because this area of memory is considered a noncacheable region of memory.
9. *Between 15–16 Mbytes*—If the address of a pending write cycle is in the 1 Mbyte of memory between 15 Mbytes and 16 Mbytes, and the WAE15M bit is set to 1, write allocate for this cycle is enabled.

10. *Write Allocate Enable 15–16 Mbytes (WAE15M)*—This condition is associated with the Write Allocate Limit mechanism and affects write allocate only if the limit specified by the WAELIM field is greater than or equal to 16 Mbytes. If the memory address is between 15 Mbytes and 16 Mbytes, and the WAE15M bit in the WHCR is set to 0, write allocate for this cycle is disabled.

## Prefetching

---

The AMD-K6 3D processor performs instruction cache prefetching for sector replacements only—as opposed to cache-line replacements. The cache prefetching results in the filling of the required cache line first, and a prefetch of the second cache line making up the other half of the sector. Furthermore, the prefetch of the second cache line is initiated only in the forward direction—that is, only if the requested cache line is the first position within the sector. From the perspective of the external bus, the two cache-line fills typically appear as two 32-byte burst read cycles occurring back-to-back or, if allowed, as pipelined cycles. The burst read cycles do not occur back-to-back (wait states occur) if the processor is not ready to start a new cycle, if higher priority data read or write requests exist, or if NA# (next address) was sampled negated. Wait states can also exist between burst cycles if the processor samples AHOLD or BOFF# asserted.

# 8

## Cache Organization

### Cache States

Table 55 shows all the possible cache-line states before and after program-generated accesses to individual cache lines. The table includes the correspondence between MESI states and writethrough or writeback states for lines in the data cache.

**Table 55. Data Cache States for Read and Write Accesses**

Type		Cache State Before Access	Access Type <sup>1</sup>	Cache State After Access	
				MESI State	Writeback Writethrough State
Cache Read	Read Miss	invalid	single read	invalid	–
		invalid	burst read <sup>2</sup> (cacheable)	shared or exclusive <sup>3</sup>	writethrough or writeback <sup>3</sup>
	Read Hit	shared	–	shared	writethrough
		exclusive	–	exclusive	writeback
		modified	–	modified	writeback
Cache Write	Write Miss	invalid	single write <sup>4</sup>	invalid	–
	Write Hit	shared	cache update and single write	shared or exclusive <sup>3</sup>	writethrough or writeback <sup>3</sup>
		exclusive or modified	cache update	modified	writeback

**Notes:**

1. Single read, single write, cache update, and writethrough = 1 to 8 bytes. Line fill = 32-byte burst read.
2. If CACHE# is driven Low and KEN# is sampled asserted.
3. If PWT is driven Low and WB/WT# is sampled High, the line is cached in the exclusive (writeback) state.
4. A write cycle occurs only if the write allocate conditions as specified in "Write Allocate" on page 240 are not met.

– Not applicable or none.

## Cache Coherency

---

Different ways exist to maintain coherency between the system memory and cache memories. Inquire cycles, internal snoops, FLUSH#, WBINVD, INVD, and line replacements all prevent inconsistencies between memories.

### Inquire Cycles

Inquire cycles are bus cycles initiated by system logic. These inquiries ensure coherency between the caches and main memory. In systems with multiple caching masters, system logic maintains cache coherency by driving inquire cycles to the processor. System logic initiates inquire cycles by asserting AHOLD, BOFF#, or HOLD to obtain control of the address bus and then driving EADS#, INV (optional), and an inquire address (A[31:5]). This type of bus cycle causes the processor to compare the tags for both its instruction and data caches with the inquire address. If there is a hit to a shared or exclusive line in the data cache or a valid line in the instruction cache, the processor asserts HIT#. If the compare hits a modified line in the data cache, the processor asserts HIT# and HITM#. If HITM# is asserted, the processor writes the modified line back to memory. If INV was sampled asserted with EADS#, a hit invalidates the line. If INV was sampled negated with EADS#, a hit leaves the line in the shared state or transitions it from the exclusive or modified to shared state.

### Internal Snooping

Internal snooping is initiated by the processor (rather than system logic) during certain cache accesses. It is used to maintain coherency between the L1 instruction and data caches.

The processor automatically snoops its instruction cache during read or write misses to its data cache, and it snoops its data cache during read misses to its instruction cache. Table 56 on page 249 summarizes the actions taken during this internal snooping.

## 8 Cache Organization

If an internal snoop hits its target, the processor does the following:

- *Data cache snoop during an instruction-cache read miss*—If modified, the line in the data cache is written back to memory. Regardless of its state, the data-cache line is invalidated and the instruction cache performs a burst cycle read from memory.
- *Instruction cache snoop during a data cache miss*—The line in the instruction cache is marked invalid, and the data-cache read or write is performed from memory.

### FLUSH#

In response to sampling FLUSH# asserted, the processor writes back any data cache lines that are in the modified state and then marks all lines in the instruction and data caches as invalid.

### WBINVD and INVD

These x86 instructions cause all cache lines to be marked as invalid. WBINVD writes back modified lines before marking all cache lines invalid. INVD does not write back modified lines.

### Cache-Line Replacement

Replacing lines in the instruction or data cache, according to the line replacement algorithms described in “Cache-Line Fills” on page 239, ensures coherency between main memory and the caches.

Table 56 on page 249 shows all possible cache-line states before and after cache snoop or invalidation operations performed with inquire cycles. This table shows all of the conditions for writethroughs and writebacks to memory.

**Table 56. Cache States for Inquiries, Snoops, Invalidation, and Replacement**

Type of Operation	Cache State Before Operation	Memory Access	Cache State After Operation		
			MESI State		Writeback Writethrough State
Inquire Cycle	shared or exclusive	-	INV=0	shared	writethrough
			INV=1	invalid	invalid
	modified	burst write (writeback)	INV=0	shared	writethrough
			INV=1	invalid	invalid
Internal Snoop	shared or exclusive	-	invalid		invalid
	modified	burst write (writeback)			
FLUSH# Signal	shared or exclusive	-	invalid		invalid
	modified	burst write (writeback)			
WBINVD Instruction	shared or exclusive	-	invalid		invalid
	modified	burst write (writeback)			
INVD Instruction	-	-	invalid		invalid
Cache-Line Replacement	shared or exclusive	-	See Table 55 on page 246		
	modified	burst write (writeback)			

Notes:  
All writebacks are 32-byte burst write cycles.  
- Not applicable or none.

# 8 Cache Organization

## Cache Snooping

Table 57 shows the conditions under which snooping occurs in the processor and the resources that are snooped.

**Table 57. Snoop Action**

Type of Event	Type of Access		Snooping Action	
			Instruction Cache	Data Cache
Inquire Cycle	System Logic		yes <sup>1</sup>	yes <sup>1</sup>
Internal Snoop	Instruction Cache	Read Miss	-	yes <sup>2</sup>
		Read Hit	-	no
	Data Cache	Read Miss	yes <sup>3</sup>	-
		Read Hit	no	-
		Write Miss	yes <sup>3</sup>	-
		Write Hit	no	-

**Notes:**

- The processor's response to an inquire cycle depends on the state of the INV input signal and the state of the cache line as follows:  
For the instruction cache, if INV is sampled negated, the line remains invalid or valid, but if INV is sampled asserted, the line is invalidated.  
For the data cache, if INV is sampled negated, valid lines remain in or transition to the shared state, a modified data cache line is written back before the line is marked shared (with HITM# asserted), and invalid lines remain invalid. For the data cache, if INV is sampled asserted, the line is marked invalid. Modified lines are written back before invalidation.
- If an internal snoop hits a modified line in the data cache, the line is written back and invalidated. Then the instruction cache performs a burst read from memory.
- If an internal snoop hits a line in the instruction cache, the instruction cache line is invalidated and the data-cache read or write is performed from memory.

- Not applicable.

## Writethrough vs. Writeback Coherency States

The terms *writethrough* and *writeback* apply to two related concepts in a read-write cache like the AMD-K6 3D processor L1 data cache. The following conditions apply to both the writethrough and writeback modes:

- **Memory Writes**—A relationship exists between external memory writes and their concurrence with cache updates:
  - An external memory write that occurs concurrently with a cache update to the same location is a writethrough. Writethroughs are driven as single cycles on the bus.
  - An external memory write that occurs after the processor has modified a cache line is a writeback. Writebacks are driven as burst cycles on the bus.
- **Coherency State**—A relationship exists between MESI coherency states and writethrough-writeback coherency states of lines in the cache as follows:
  - Shared MESI lines are in the writethrough state.
  - Modified and exclusive MESI lines are in the writeback state.

## A20M# Masking of Cache Accesses

Although the processor samples A20M# as a level-sensitive input on every clock edge, it should only be asserted in Real mode. The processor applies the A20M# masking to its tags, through which all programs access the caches. Therefore, assertion of A20M# affects all addresses (cache and external memory), including the following:

- Cache-line fills (caused by read misses)
- Cache writethroughs (caused by write misses or write hits to lines in the shared state)

However, A20M# does not mask writebacks or invalidations caused by the following actions:

- Internal snoops
- Inquire cycles
- The FLUSH# signal
- The WBINVD instruction



## **8** *Cache Organization*

---

## **Floating-Point and Multimedia Execution Units**

### **Floating-Point Execution Unit**

---

The AMD-K6 3D processor contains an IEEE 754-compatible and 854-compatible floating-point execution unit designed to accelerate the performance of software that utilizes the x86 floating-point instruction set. Floating-point software is typically written to manipulate numbers that are very large or very small, that require a high degree of precision, or that result from complex mathematical operations such as transcendentals. Applications that take advantage of floating-point operations include geometric calculations for graphics acceleration, scientific, statistical, and engineering applications, and business applications that use large amounts of high-precision data.

The high-performance floating-point execution unit contains an adder unit, a multiplier unit, and a divide/square root unit. These low-latency units can execute floating-point instructions in as few as two processor clocks. To increase performance, the processor is designed to simultaneously decode most floating-point instructions with most short-decodeable instructions.

See Chapter 3, “Software Environment” on page 23 for a description of the floating-point data types, registers, and instructions.

## **9** *Floating-Point and Multimedia Execution Units*

### **Handling Floating-Point Exceptions**

The processor provides the following two types of exception handling for floating-point exceptions:

- If the numeric error (NE) bit in CR0 is set to 1, the processor invokes the interrupt 10h handler. In this manner, the floating-point exception is completely handled by software.
- If the NE bit in CR0 is set to 0, the processor requires external logic to generate an interrupt on the INTR signal in order to handle the exception.

### **External Logic Support of Floating-Point Exceptions**

The processor provides the FERR# (Floating-Point Error) and IGNNE# (Ignore Numeric Error) signals to allow the external logic to generate the interrupt in a manner consistent with IBM-compatible PC/AT systems. The assertion of FERR# indicates the occurrence of an unmasked floating-point exception resulting from the execution of a floating-point instruction. IGNNE# is used by the external hardware to control the effect of an unmasked floating-point exception. Under certain circumstances, if IGNNE# is sampled asserted, the processor ignores the floating-point exception.

Figure 80 on page 255 illustrates an implementation of external logic for supporting floating-point exceptions. The following example explains the operation of the external logic in Figure 80:

As the result of a floating-point exception, the processor asserts FERR#. The assertion of FERR# and the sampling of IGNNE# negated indicates the processor has stopped instruction execution and is waiting for an interrupt. The assertion of FERR# leads to the assertion of INTR by the interrupt controller. The processor acknowledges the interrupt and jumps to the corresponding interrupt service routine in which an I/O write cycle to address port F0h leads to the assertion of IGNNE#. When IGNNE# is sampled asserted, the processor ignores the floating-point exception and continues instruction execution. When the processor negates FERR#, the external logic negates IGNNE#.

See “FERR# (Floating-Point Error)” on page 158 and “IGNNE# (Ignore Numeric Exception)” on page 163 for more details.

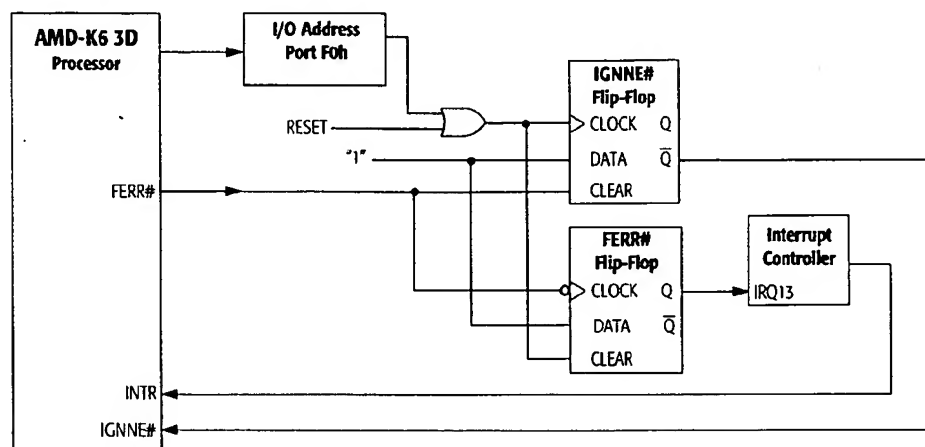


Figure 80. External Logic for Supporting Floating-Point Exceptions

## Multimedia and 3D Execution Units

The multimedia and 3D execution units of the processor are designed to accelerate the performance of software written using the industry-standard MMX instructions and the new 3D instructions. Applications that can take advantage of the MMX and 3D instructions include graphics, video and audio compression and decompression, speech recognition, and telephony applications.

The MMX multimedia execution unit can execute MMX instructions in a single processor clock. All MMX and 3D arithmetic instructions are pipelined for higher performance. To increase performance, the processor is designed to simultaneously decode all MMX and 3D instructions with most other instructions.

For more information on MMX instructions, see Appendix A, "MMX Multimedia Technology" on page 347. For more information on 3D instructions, see Chapter 4, "3D Technology" on page 81.

---

## **9** *Floating-Point and Multimedia Execution Units*

---

### **Floating-Point and MMX/3D Instruction Compatibility**

---

#### **Registers**

The eight 64-bit MMX registers (which are also utilized by 3D instructions) are mapped on the floating-point stack. This enables backward compatibility with all existing software. For example, the register saving event that is performed by operating systems during task switching requires no changes to the operating system. The same support provided in an operating system's interrupt 7 handler (Device Not Available) for saving and restoring the floating-point registers also supports saving and restoring the MMX registers.

#### **Exceptions**

There are no new exceptions defined for supporting the MMX and 3D instructions. All exceptions that occur while decoding or executing an MMX or 3D instruction are handled in existing exception handlers without modification. See "3D Exceptions" on page 93 for more information.

#### **FERR# and IGNNE#**

MMX instructions and 3D instructions do not generate floating-point exceptions. However, if an unmasked floating-point exception is pending, the processor asserts FERR# at the instruction boundary of the next floating-point instruction, MMX instruction, 3D instruction or WAIT instruction.

The sampling of IGNNE# asserted only affects processor operation during the execution of an error-sensitive floating-point instruction, MMX instruction, 3D instruction or WAIT instruction when the NE bit in CR0 is set to 0.

# 10

## System Management Mode (SMM)

### Overview

---

SMM is an alternate operating mode entered by way of a system management interrupt (SMI#) and handled by an interrupt service routine. SMM is designed for system control activities such as power management. These activities appear transparent to conventional operating systems like DOS and Windows. SMM is primarily targeted for use by the Basic Input Output System (BIOS) and specialized low-level device drivers. The code and data for SMM are stored in the SMM memory area, which is isolated from main memory.

The processor enters SMM by the system logic's assertion of the SMI# interrupt and the processor's acknowledgment by the assertion of SMIACK#. At this point the processor saves its state into the SMM memory state-save area and jumps to the SMM service routine. The processor returns from SMM when it executes the RSM (resume) instruction from within the SMM service routine. Subsequently, the processor restores its state from the SMM save area, negates SMIACK#, and resumes execution with the instruction following the point where it entered SMM.

The following sections summarize the SMM state-save area, entry into and exit from SMM, exceptions and interrupts in SMM, memory allocation and addressing in SMM, and the SMI# and SMIACK# signals.

# **10** *System Management Mode (SMM)*

---

## **SMM Operating Mode and Default Register Values**

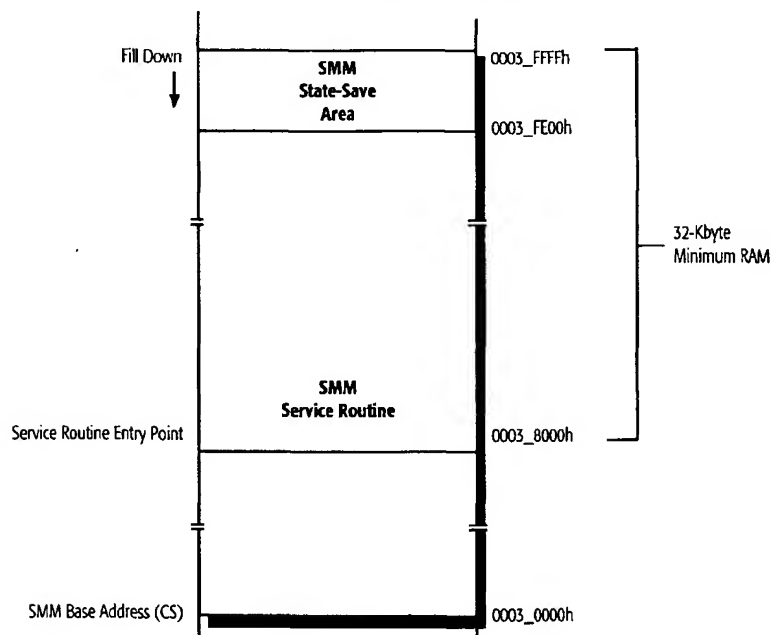
---

The software environment within SMM has the following characteristics:

- Addressing and operation in Real mode
- 4-Gbyte segment limits
- Default 16-bit operand, address, and stack sizes, although instruction prefixes can override these defaults
- Control transfers that do not override the default operand size truncate the EIP to 16 bits
- Far jumps or calls cannot transfer control to a segment with a base address requiring more than 20 bits, as in Real mode segment-base addressing
- A20M# is masked
- Interrupt vectors use the Real-mode interrupt vector table
- The IF flag in EFLAGS is cleared (INTR not recognized)
- The TF flag in EFLAGS is cleared
- The NMI and INIT interrupts are disabled
- Debug register DR7 is cleared (debug traps disabled)

Figure 81 on page 259 shows the default map of the SMM memory area. It consists of a 64-Kbyte area, between 0003\_0000h and 0003\_FFFFh, of which the top 32 Kbytes (0003\_8000h to 0003\_FFFFh) must be populated with RAM. The default code-segment (CS) base address for the area—called the SMM base address—is at 0003\_0000h. The top 512 bytes (0003\_FE00h to 0003\_FFFFh) contain a fill-down SMM state-save area. The default entry point for the SMM service routine is 0003\_8000h.

# System Management Mode (SMM) 10



**Figure 81. SMM Memory**

Table 58 shows the initial state of registers when entering SMM.

**Table 58. Initial State of Registers in SMM**

Registers	SMM Initial State
General Purpose Registers	unmodified
EFLAGS	0000_0002h
CR0	PE, EM, TS, and PG are cleared (bits 0, 2, 3, and 31). The other bits are unmodified.
DR7	0000_0400h
GDTR, LDTR, IDTR, TSSR, DR6	unmodified
EIP	0000_8000h
CS	0003_0000h
DS, ES, FS, GS, SS	0000_0000h



# 10 System Management Mode (SMM)

## SMM State-Save Area

When the processor acknowledges an SMI# interrupt by asserting SMIACK#, it saves its state in a 512-byte SMM state-save area shown in Table 59. The save begins at the top of the SMM memory area (SMM base address + FFFFh) and fills down to SMM base address + FE00h.

Table 59 shows the offsets in the SMM state-save area relative to the SMM base address. The SMM service routine can alter any of the read/write values in the state-save area.

Table 59. SMM State-Save Area Map

Address Offset	Contents Saved
FFFC	CR0
FFF8	CR3
FFF4	EFLAGS
FFF0	EIP
FFEC	EDI
FFE8	ESI
FFE4	EBP
FFE0	ESP
FFDC	EBX
FFD8	EDX
FFD4	ECX
FFD0	EAX
FFC8	DR6
FFC4	DR7
FFC0	TR
FFB8	LDTR Base
FFB4	GS
FFB0	FS
FFAC	DS
FFA8	SS
FFA4	CS

**Notes:**

- No data dump at that address
- \* Only contains information if SMI# is asserted during a valid I/O bus cycle.

# System Management Mode (SMM) 10

**Table 59. SMM State-Save Area Map (continued)**

Address Offset	Contents Saved
FFA8h	ES
FFA4h	I/O Trap Dword
FFA0h	—
FF9Ch	I/O Trap EIP*
FF98h	—
FF94h	—
FF90h	IDT Base
FF8Ch	IDT Limit
FF88h	GDT Base
FF84h	GDT Limit
FF80h	TSS Attr
FF7Ch	TSS Base
FF78h	TSS Limit
FF74h	—
FF70h	LDT High
FF6Ch	LDT Low
FF68h	GS Attr
FF64h	GS Base
FF60h	GS Limit
FF5Ch	FS Attr
FF58h	FS Base
FF54h	FS Limit
FF50h	DS Attr
FF4Ch	DS Base
FF48h	DS Limit
FF44h	SS Attr
FF40h	SS Base
FF3Ch	SS Limit
FF38h	CS Attr
FF34h	CS Base
FF30h	CS Limit
<b>Notes:</b> — No data dump at that address * Only contains information if SMI# is asserted during a valid I/O bus cycle.	

# 10 System Management Mode (SMM)

Table 59. SMM State-Save Area Map (continued)

Address Offset	Contents Saved
FF2Ch	ES Attr
FF28h	ES Base
FF24h	ES Limit
FF20h	—
FF1Ch	—
FF18h	—
FF14h	CR2
FF10h	CR4
FF0Ch	I/O Restart ESI*
FF08h	I/O Restart ECX*
FF04h	I/O Restart EDI*
FF02h	HALT Restart Slot
FF00h	I/O Trap Restart Slot
FEFCh	SMM RevID
FEF8h	SMM Base
FEF7h–FE00h	—
<b>Notes:</b> — No data dump at that address * Only contains information if SM <sup>1</sup> # is asserted during a valid I/O bus cycle.	

## SMM Revision Identifier

The SMM revision identifier at offset FEFCh in the SMM state-save area specifies the version of SMM and the extensions that are available on the processor. The SMM revision identifier fields are as follows:

- Bits 31–18—Reserved
- Bit 17—SMM base address relocation (1 = enabled)
- Bit 16—I/O trap restart (1 = enabled)
- Bits 15–0—SMM revision level for the AMD-K6 3D processor = 0002h

Table 60 shows the format of the SMM Revision Identifier.

**Table 60. SMM Revision Identifier**

31–18	17	16	15–0
Reserved	SMM Base Relocation	I/O Trap Extension	SMM Revision Level
0	1	1	0002h

## SMM Base Address

During RESET, the processor sets the base address of the code-segment (CS) for the SMM memory area—the SMM base address—to its default, 0003\_0000h. The SMM base address at offset FEF8h in the SMM state-save area can be changed by the SMM service routine to any address that is aligned to a 32-Kbyte boundary. (Locations not aligned to a 32-Kbyte boundary cause the processor to enter the Shutdown state when executing the RSM instruction.)

In some operating environments it may be desirable to relocate the 64-Kbyte SMM memory area to a high memory area in order to provide more low memory for legacy software. During system initialization, the base of the 64-Kbyte SMM memory area is relocated by the BIOS. To relocate the SMM base address, the system enters the SMM handler at the default address. This handler changes the SMM base address location in the SMM state-save area, copies the SMM handler to the new location, and exits SMM.

The next time SMM is entered, the processor saves its state at the new base address. This new address is used for every SMM entry until the SMM base address in the SMM state-save area is changed or a hardware reset occurs.

# **10** *System Management Mode (SMM)*

---

## **Halt Restart Slot**

---

During entry into SMM, the halt restart slot at offset FF02h in the SMM state-save area indicates if SMM was entered from the Halt state. Before returning from SMM, the halt restart slot (offset FF02h) can be written to by the SMM service routine to specify whether the return from SMM takes the processor back to the Halt state or to the next instruction after the HLT instruction.

Upon entry into SMM, the halt restart slot is defined as follows:

- *Bits 15–1*—Reserved
- *Bit 0*—Point of entry to SMM:
  - 1 = entered from Halt state
  - 0 = not entered from Halt state

After entry into the SMI handler and before returning from SMM, the halt restart slot can be written using the following definition:

- *Bits 15–1*—Reserved
- *Bit 0*—Point of return when exiting from SMM:
  - 1 = return to Halt state
  - 0 = return to next instruction after the HLT instruction

If the return from SMM takes the processor back to the Halt state, the HLT instruction is not re-executed, but the Halt special bus cycle is driven on the bus after the return.

## I/O Trap Dword

If the assertion of SMI# is recognized during the execution of an I/O instruction, the I/O trap dword at offset FFA4h in the SMM state-save area contains information about the instruction. The fields of the I/O trap dword are configured as follows:

- Bits 31–16—I/O port address
- Bits 15–4—Reserved
- Bit 3—REP (repeat) string operation (1 = REP string, 0 = not a REP string)
- Bit 2—I/O string operation (1 = I/O string, 0 = not an I/O string)
- Bit 1—Valid I/O instruction (1 = valid, 0 = invalid)
- Bit 0—Input or output instruction (1 = INx, 0 = OUTx)

Table 61 shows the format of the I/O trap dword.

**Table 61. I/O Trap Dword Configuration**

31–16	15–4	3	2	1	0
I/O Port Address	Reserved	REP String Operation	I/O String Operation	Valid I/O Instruction	Input or Output

The I/O trap dword is related to the I/O trap restart slot (see “I/O Trap Restart Slot” on page 266). If bit 1 of the I/O trap dword is set by the processor, it means that SMI# was asserted during the execution of an I/O instruction. The SMI handler tests bit 1 to see if there is a valid I/O instruction trapped. If the I/O instruction is valid, the SMI handler is required to ensure the I/O trap restart slot is set properly. The I/O trap restart slot informs the processor whether it should re-execute the I/O instruction after the RSM or execute the instruction following the trapped I/O instruction.

*Note: If SMI# is sampled asserted during an I/O bus cycle a minimum of three clock edges before BRDY# is sampled asserted, the associated I/O instruction is guaranteed to be trapped by the SMI handler.*

# 10 System Management Mode (SMM)

## I/O Trap Restart Slot

The I/O trap restart slot at offset FF00h in the SMM state-save area specifies whether the trapped I/O instruction should be re-executed on return from SMM. This slot in the state-save area is called the *I/O instruction restart* function. Re-executing a trapped I/O instruction is useful, for example, if an I/O write occurs to a disk that is powered down. The system logic monitoring such an access can assert SMI#. Then the SMM service routine would query the system logic, detect a failed I/O write, take action to power-up the I/O device, enable the I/O trap restart slot feature, and return from SMM.

The fields of the I/O trap restart slot are defined as follows:

- **Bits 31–16**—Reserved
- **Bits 15–0**—I/O instruction restart on return from SMM:
  - 0000h = execute the next instruction after the trapped I/O instruction
  - 00FFh = re-execute the trapped I/O instruction

Table 62 shows the format of the I/O trap restart slot.

**Table 62. I/O Trap Restart Slot**

31–16	15–0
Reserved	I/O instruction restart on return from SMM: <ul style="list-style-type: none"><li>■ 0000h = execute the next instruction after the trapped I/O</li><li>■ 00FFh = re-execute the trapped I/O instruction</li></ul>

The processor initializes the I/O trap restart slot to 0000h upon entry into SMM. If SMM was entered due to a trapped I/O instruction, the processor indicates the validity of the I/O instruction by setting or clearing bit 1 of the I/O trap dword at offset FFA4h in the SMM state-save area. The SMM service routine should test bit 1 of the I/O trap dword to determine if a valid I/O instruction was being executed when entering SMM and before writing the I/O trap restart slot. If the I/O instruction is valid, the SMM service routine can safely rewrite the I/O trap restart slot with the value 00FFh, which causes the processor to re-execute the trapped I/O instruction when the RSM instruction is executed. If the I/O instruction is invalid, writing the I/O trap restart slot has undefined results.

If a second SMI# is asserted and a valid I/O instruction was trapped by the first SMM handler, the processor services the second SMI# prior to re-executing the trapped I/O instruction. The second entry into SMM never has bit 1 of the I/O trap dword set, and the second SMM service routine must not rewrite the I/O trap restart slot.

During a simultaneous SMI# I/O instruction trap and debug breakpoint trap, the AMD-K6 3D processor first responds to the SMI# and postpones recognizing the debug exception until after returning from SMM via the RSM instruction. If the debug registers DR3–DR0 are used while in SMM, they must be saved and restored by the SMM handler. The processor automatically saves and restores DR7–DR6. If the I/O trap restart slot in the SMM state-save area contains the value 00FFh when the RSM instruction is executed, the debug trap does not occur until after the I/O instruction is re-executed.

## **Exceptions, Interrupts, and Debug in SMM**

---

During an SMI# I/O trap, the exception/interrupt priority of the processor changes from its normal priority. The normal priority places the debug traps at a priority higher than the sampling of the FLUSH# or SMI# signals. However, during an SMI# I/O trap, the sampling of the FLUSH# or SMI# signals takes precedence over debug traps.

The processor recognizes the assertion of NMI within SMM immediately after the completion of an IRET instruction. Once NMI is recognized within SMM, NMI recognition remains enabled until SMM is exited, at which point NMI masking is restored to the state it was in before entering SMM.



---

## **10** *System Management Mode (SMM)*

---

# 11

## Test and Debug

The AMD-K6 3D processor implements various test and debug modes to enable the functional and manufacturing testing of systems and boards that use the processor. In addition, the debug features of the processor allow designers to debug the instruction execution of software components. This chapter describes the following test and debug features:

- *Built-In Self-Test (BIST)*—The BIST, which is invoked after the falling transition of RESET, runs internal tests that exercise most on-chip RAM structures.
- *Tri-State Test Mode*—A test mode that causes the processor to float its output and bidirectional pins.
- *Boundary-Scan Test Access Port (TAP)*—The Joint Test Action Group (JTAG) test access function defined by the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1-1990)* specification.
- *Level-One (L1) Cache Inhibit*—A feature that disables the processor's internal L1 instruction and data caches.
- *Debug Support*—Consists of all x86-compatible software debug features, including the debug extensions.

# **11** *Test and Debug*

---

## **Built-In Self-Test (BIST)**

---

Following the falling transition of RESET, the processor unconditionally runs its BIST. The internal resources tested during BIST include the following:

- L1 instruction and data caches
- Instruction and Data Translation Lookaside Buffers (TLBs)

The contents of the EAX general-purpose register after the completion of reset indicate if the BIST was successful. If EAX contains 0000\_0000h, then BIST was successful. If EAX is non-zero, the BIST failed. Following the completion of the BIST, the processor jumps to address FFFF\_FFF0h to start instruction execution, regardless of the outcome of the BIST.

The BIST takes approximately 295,000 processor clocks to complete.

## **Tri-State Test Mode**

---

The Tri-State Test mode causes the processor to float its output and bidirectional pins, which is useful for board-level manufacturing testing. In this mode, the processor is electrically isolated from other components on a system board, allowing automated test equipment (ATE) to test components that drive the same signals as those the processor floats.

If the FLUSH# signal is sampled Low during the falling transition of RESET, the processor enters the Tri-State Test mode. (See “FLUSH# (Cache Flush)” on page 159 for the specific sampling requirements.) The signals floated in the Tri-State Test mode are as follows:

- |            |           |           |
|------------|-----------|-----------|
| ■ A[31:3]  | ■ D/C#    | ■ M/IO#   |
| ■ ADS#     | ■ D[63:0] | ■ PCD     |
| ■ ADSC#    | ■ DP[7:0] | ■ PCHK#   |
| ■ AP       | ■ FERR#   | ■ PWT     |
| ■ APCHK#   | ■ HIT#    | ■ SCYC    |
| ■ BE[7:0]# | ■ HITM#   | ■ SMIACK# |
| ■ BREQ     | ■ HLDA    | ■ W/R#    |
| ■ CACHE#   | ■ LOCK#   |           |

The VCC2DET, VCC2H/L#, and TDO signals are the only outputs not floated in the Tri-State Test mode. VCC2DET and VCC2H/L# must remain Low to ensure the system continues to supply the specified processor core voltage to the V<sub>CC2</sub> pins. TDO is never floated because the Boundary-Scan Test Access Port must remain enabled at all times, including during the Tri-State Test mode.

The Tri-State Test mode is exited when the processor samples RESET asserted.

## **Boundary-Scan Test Access Port (TAP)**

---

The boundary-scan Test Access Port (TAP) is an IEEE standard that defines synchronous scanning test methods for complex logic circuits, such as boards containing a processor. The AMD-K6 3D processor supports the TAP standard defined in the *IEEE Standard Test Access Port and Boundary-Scan Architecture (IEEE 1149.1-1990)* specification.

Boundary scan testing uses a shift register consisting of the serial interconnection of boundary-scan cells that correspond to each I/O buffer of the processor. This non-inverting register chain, called a Boundary Scan Register (BSR), can be used to capture the state of every processor pin and to drive every processor output and bidirectional pin to a known state.

Each BSR of every component on a board that implements the boundary-scan architecture can be serially interconnected to enable component interconnect testing.

### **Test Access Port**

The TAP consists of the following:

- *Test Access Port (TAP) Controller*—The TAP controller is a synchronous, finite state machine that uses the TMS and TDI input signals to control a sequence of test operations. See “TAP Controller State Machine” on page 278 for a list of TAP states and their definition.
- *Instruction Register (IR)*—The IR contains the instructions that select the test operation to be performed and the Test

# 11 Test and Debug

Data Register (TDR) to be selected. See “TAP Registers” on page 273 for more details on the IR.

- *Test Data Registers (TDR)*—The three TDRs are used to process the test data. Each TDR is selected by an instruction in the Instruction Register (IR). See “TAP Registers” on page 273 for a list of these registers and their functions.

## TAP Signals

The test signals associated with the TAP controller are as follows:

- *TCK*—The Test Clock for all TAP operations. The rising edge of TCK is used for sampling TAP signals, and the falling edge of TCK is used for asserting TAP signals. The state of the TMS signal sampled on the rising edge of TCK causes the state transitions of the TAP controller to occur. TCK can be stopped in the logic 0 or 1 state.
- *TDI*—The Test Data Input represents the input to the most significant bit of all TAP registers, including the IR and all test data registers. Test data and instructions are serially shifted by one bit into their respective registers on the rising edge of TCK.
- *TDO*—The Test Data Output represents the output of the least significant bit of all TAP registers, including the IR and all test data registers. Test data and instructions are serially shifted by one bit out of their respective registers on the falling edge of TCK.
- *TMS*—The Test Mode Select input specifies the test function and sequence of state changes for boundary-scan testing. If TMS is sampled High for five or more consecutive clocks, the TAP controller enters its reset state.
- *TRST#*—The Test Reset signal is an asynchronous reset that unconditionally causes the TAP controller to enter its reset state.

Refer to Chapter 14, “Electrical Data” on page 303 and Chapter 16, “Signal Switching Characteristics” on page 313 to obtain the electrical specifications of the test signals.

## TAP Registers

The processor provides an Instruction Register (IR) and three Test Data Registers (TDR) to support the boundary-scan architecture. The IR and one of the TDRs—the Boundary-Scan Register (BSR)—consist of a shift register and an output register. The shift register is loaded in parallel in the Capture states. (See “TAP Controller State Machine” on page 278 for a description of the TAP controller states.) In addition, the shift register is loaded and shifted serially in the Shift states. The output register is loaded in parallel from its corresponding shift register in the Update states.

### Instruction Register (IR)

The IR is a 5-bit register, without parity, that determines which instruction to run and which test data register to select. When the TAP controller enters the Capture-IR state, the processor loads the following bits into the IR shift register:

- *01b*—Loaded into the two least significant bits, as specified by the IEEE 1149.1 standard
- *000b*—Loaded into the three most significant bits

Loading 00001b into the IR shift register during the Capture-IR state results in loading the SAMPLE/PRELOAD instruction.

For each entry into the Shift-IR state, the IR shift register is serially shifted by one bit toward the TDO pin. During the shift, the most significant bit of the IR shift register is loaded from the TDI pin.

The IR output register is loaded from the IR shift register in the Update-IR state, and the current instruction is defined by the IR output register. See “TAP Instructions” on page 277 for a list and definition of the instructions supported by the processor.

### Boundary Scan Register (BSR)

The BSR is a Test Data Register consisting of the interconnection of 152 boundary-scan cells. Each output and bidirectional pin of the processor requires a two-bit cell, where one bit corresponds to the pin and the other bit is the output enable for the pin. When a 0 is shifted into the enable bit of a cell, the corresponding pin is floated, and when a 1 is shifted into the enable bit, the pin is driven valid. Each input pin requires a one-bit cell that corresponds to the pin. The last cell of the BSR is reserved and does not correspond to any processor pin.

# 11 Test and Debug

The total number of bits that comprise the BSR is 281. Table 63 on page 275 lists the order of these bits, where TDI is the input to bit 280, and TDO is driven from the output of bit 0. The entries listed as *pin\_E* (where *pin* is an output or bidirectional signal) are the enable bits.

If the BSR is the register selected by the current instruction and the TAP controller is in the Capture-DR state, the processor loads the BSR shift register as follows:

- If the current instruction is SAMPLE/PRELOAD, then the current state of each input, output, and bidirectional pin is loaded. A bidirectional pin is treated as an output if its enable bit equals 1, and it is treated as an input if its enable bit equals 0.
- If the current instruction is EXTEST, then the current state of each input pin is loaded. A bidirectional pin is treated as an input, regardless of the state of its enable.

While in the Shift-DR state, the BSR shift register is serially shifted toward the TDO pin. During the shift, bit 280 of the BSR is loaded from the TDI pin.

The BSR output register is loaded with the contents of the BSR shift register in the Update-DR state. If the current instruction is EXTEST, the processor's output pins, as well as those bidirectional pins that are enabled as outputs, are driven with their corresponding values from the BSR output register.

**Table 63. Boundary Scan Bit Definitions**

Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable
280	D35_E	247	D21	214	D4_E	181	A3	148	A20	115	A16
279	D35	246	D18_E	213	D4	180	A31_E	147	A13_E	114	FERR_E
278	D29_E	245	D18	212	DP0_E	179	A31	146	A13	113	FERR#
277	D29	244	D19_E	211	DP0	178	A21_E	145	DP7_E	112	HIT_E
276	D33_E	243	D19	210	HOLD	177	A21	144	DP7	111	HIT#
275	D33	242	D16_E	209	BOFF#	176	A30_E	143	BE6_E	110	BE7_E
274	D27_E	241	D16	208	AHOLD	175	A30	142	BE6#	109	BE7#
273	D27	240	D17_E	207	STPCLK#	174	A7_E	141	A12_E	108	NA#
272	DP3_E	239	D17	206	INIT	173	A7	140	A12	107	ADSC_E
271	DP3	238	D15_E	205	IGNNE#	172	A24_E	139	CLK	106	ADSC#
270	D25_E	237	D15	204	BF1	171	A24	138	BE4_E	105	BE5_E
269	D25	236	DP1_E	203	BF2	170	A18_E	137	BE4#	104	BE5#
268	D0_E	235	DP1	202	RESET	169	A18	136	A10_E	103	WB/WT#
267	D0	234	D13_E	201	BF0	168	A5_E	135	A10	102	PWT_E
266	D30_E	233	D13	200	FLUSH#	167	A5	134	D63_E	101	PWT
265	D30	232	D6_E	199	INTR	166	A22_E	133	D63	100	BE3_E
264	DP2_E	231	D6	198	NMI	165	A22	132	BE2_E	99	BE3#
263	DP2	230	D14_E	197	SMI#	164	EADS#	131	BE2#	98	BREQ_E
262	D2_E	229	D14	196	A25_E	163	A4_E	130	A15_E	97	BREQ
261	D2	228	D11_E	195	A25	162	A4	129	A15	96	PCD_E
260	D28_E	227	D11	194	A23_E	161	HITM_E	128	BRDY#	95	PCD
259	D28	226	D1_E	193	A23	160	HITM#	127	BE1_E	94	WR_E
258	D24_E	225	D1	192	A26_E	159	A9_E	126	BE1#	93	W/R#
257	D24	224	D12_E	191	A26	158	A9	125	A14_E	92	SMIACT_E
256	D26_E	223	D12	190	A29_E	157	SCYC_E	124	A14	91	SMIACT#
255	D26	222	D10_E	189	A29	156	SCYC	123	BRDYC#	90	EWBE#
254	D22_E	221	D10	188	A28_E	155	A8_E	122	BE0_E	89	DC_E
253	D22	220	D7_E	187	A28	154	A8	121	BE0#	88	D/C#
252	D23_E	219	D7	186	A27_E	153	A19_E	120	A17_E	87	APCHK_E
251	D23	218	D8_E	185	A27	152	A19	119	A17	86	APCHK#
250	D20_E	217	D8	184	A11_E	151	A6_E	118	KEN#	85	CACHE_E
249	D20	216	D9_E	183	A11	150	A6	117	A20M#	84	CACHE#
248	D21_E	215	D9	182	A3_E	149	A20_E	116	A16_E	83	ADS_E



# 11 Test and Debug

**Table 63. Boundary Scan Bit Definitions (continued)**

Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable	Bit	Pin/Enable
82	ADS#	68	DP6_E	54	D53_E	40	D43_E	26	D38_E	12	D3_E
81	AP_E	67	DP6	53	D53	39	D43	25	D38	11	D3
80	AP	66	D54_E	52	D47_E	38	D62_E	24	D58_E	10	D39_E
79	INV	65	D54	51	D47	37	D62	23	D58	9	D39
78	HLDA_E	64	D50_E	50	D59_E	36	D49_E	22	D42_E	8	D32_E
77	HLDA	63	D50	49	D59	35	D49	21	D42	7	D32
76	PCHK_E	62	D56_E	48	D51_E	34	DP4_E	20	D36_E	6	D5_E
75	PCHK#	61	D56	47	D51	33	DP4	19	D36	5	D5
74	LOCK_E	60	D55_E	46	D45_E	32	D46_E	18	D60_E	4	D37_E
73	LOCK#	59	D55	45	D45	31	D46	17	D60	3	D37
72	MIO_E	58	D48_E	44	D61_E	30	D41_E	16	D40_E	2	D31_E
71	M/IO#	57	D48	43	D61	29	D41	15	D40	1	D31
70	D52_E	56	D57_E	42	DP5_E	28	D44_E	14	D34_E	0	Reserved
69	D52	55	D57	41	DP5	27	D44	13	D34		

## Device Identification Register (DIR)

The DIR is a 32-bit Test Data Register selected during the execution of the IDCODE instruction. The fields of the DIR and their values are shown in Table 64 and are defined as follows:

- *Version Code*—This 4-bit field is incremented by AMD manufacturing for each major revision of silicon.
- *Part Number*—This 16-bit field identifies the specific processor model.
- *Manufacturer*—This 11-bit field identifies the manufacturer of the component (AMD).
- *LSB*—The least significant bit (LSB) of the DIR is always set to 1, as specified by the IEEE 1149.1 standard.

**Table 64. Device Identification Register**

Version Code (Bits 31–28)	Part Number (Bits 27–12)	Manufacturer (Bits 11–1)	LSB (Bit 0)
Xh	0580h	00000000001b	1b

## Bypass Register (BR)

The BR is a Test Data Register consisting of a 1-bit shift register that provides the shortest path between TDI and TDO. When the processor is not involved in a test operation, the BR can be selected by an instruction to allow the transfer of test data through the processor without having to serially scan the test data through the BSR. This functionality preserves the state of the BSR and significantly reduces test time.

The BR register is selected by the **BYPASS** and **HIGHZ** instructions as well as by any instructions not supported by the processor.

## TAP Instructions

The processor supports the three instructions required by the IEEE 1149.1 standard—**EXTEST**, **SAMPLE/PRELOAD**, and **BYPASS**—as well as two additional optional instructions—**IDCODE** and **HIGHZ**.

Table 65 shows the complete set of TAP instructions supported by the processor along with the 5-bit Instruction Register encoding and the register selected by each instruction.

**Table 65. Supported Tap Instructions**

Instruction	Encoding	Register	Description
EXTEST <sup>1</sup>	00000b	BSR	Sample inputs and drive outputs
SAMPLE / PRELOAD	00001b	BSR	Sample inputs and outputs, then load the BSR
IDCODE	00010b	DIR	Read DIR
HIGHZ	00011b	BR	Float outputs and bidirectional pins
BYPASS <sup>2</sup>	00100b–11110b	BR	Undefined instruction, execute the BYPASS instruction
BYPASS <sup>3</sup>	11111b	BR	Connect TDI to TDO to bypass the BSR
<b>Notes:</b> <ol style="list-style-type: none"> <li>Following the execution of the EXTEST instruction, the processor must be reset in order to return to normal, non-test operation.</li> <li>These instruction encodings are undefined on the AMD-K6 3D processor and default to the BYPASS instruction.</li> <li>Because the TDI input contains an internal pullup, the BYPASS instruction is executed if the TDI input is not connected or open during an instruction scan operation. The BYPASS instruction does not affect the normal operational state of the processor.</li> </ol>			

## EXTEST

When the EXTEST instruction is executed, the processor loads the BSR shift register with the current state of the input and bidirectional pins in the Capture-DR state and drives the output and bidirectional pins with the corresponding values from the BSR output register in the Update-DR state.

# 11 *Test and Debug*

---

## **SAMPLE/PRELOAD**

The SAMPLE/PRELOAD instruction performs two functions. These functions are as follows:

- During the Capture-DR state, the processor loads the BSR shift register with the current state of every input, output, and bidirectional pin.
- During the Update-DR state, the BSR output register is loaded from the BSR shift register in preparation for the next EXTEST instruction.

The SAMPLE/PRELOAD instruction does not affect the normal operational state of the processor.

## **BYPASS**

The BYPASS instruction selects the BR register, which reduces the boundary-scan length through the processor from 281 to one (TDI to BR to TDO). The BYPASS instruction does not affect the normal operational state of the processor.

## **IDCODE**

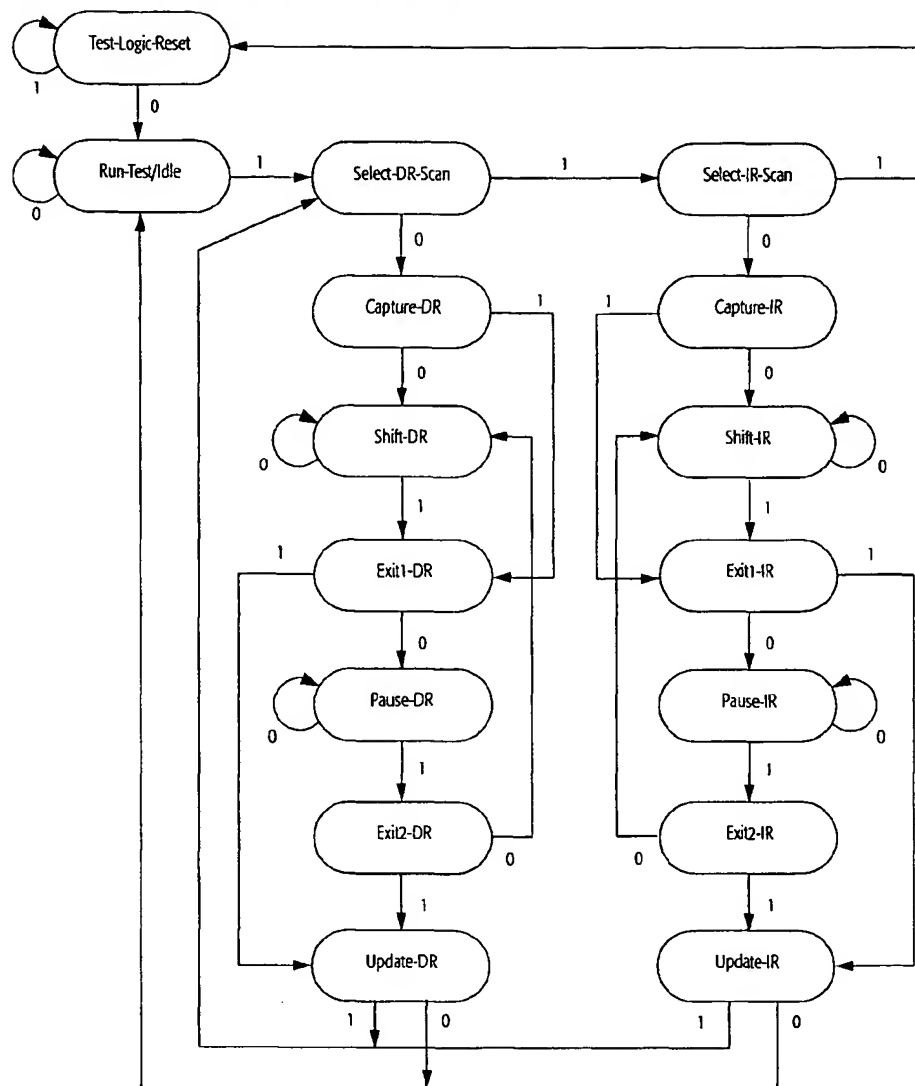
The IDCODE instruction selects the DIR register, allowing the device identification code to be shifted out of the processor. This instruction is loaded into the IR when the TAP controller is reset. The IDCODE instruction does not affect the normal operational state of the processor.

## **HIGHZ**

The HIGHZ instruction forces all output and bidirectional pins to be floated. During this instruction, the BR is selected and the normal operational state of the processor is not affected.

## **TAP Controller State Machine**

The TAP controller state diagram is shown in Figure 82 on page 279. State transitions occur on the rising edge of TCK. The logic 0 or 1 next to the states represents the value of the TMS signal sampled by the processor on the rising edge of TCK.



IEEE Std 1149.1-1990, Copyright © 1990. IEEE. All rights reserved

**Figure 82. TAP State Diagram**

# 11 *Test and Debug*

---

The states of the TAP controller are described as follows:

**Test-Logic-Reset**

This state represents the initial reset state of the TAP controller and is entered when the processor samples RESET asserted, when TRST# is asynchronously asserted, and when TMS is sampled High for five or more consecutive clocks. In addition, this state can be entered from the Select-IR-Scan state. The IR is initialized with the IDCODE instruction, and the processor's normal operation is not affected in this state.

**Capture-DR**

During the SAMPLE/PRELOAD instruction, the processor loads the BSR shift register with the current state of every input, output, and bidirectional pin. During the EXTEST instruction, the processor loads the BSR shift register with the current state of every input and bidirectional pin.

**Capture-IR**

When the TAP controller enters the Capture-IR state, the processor loads 01b into the two least significant bits of the IR shift register and loads 000b into the three most significant bits of the IR shift register.

**Shift-DR**

While in the Shift-DR state, the selected TDR shift register is serially shifted toward the TDO pin. During the shift, the most significant bit of the TDR is loaded from the TDI pin.

**Shift-IR**

While in the Shift-IR state, the IR shift register is serially shifted toward the TDO pin. During the shift, the most significant bit of the IR is loaded from the TDI pin.

**Update-DR**

During the SAMPLE/PRELOAD instruction, the BSR output register is loaded with the contents of the BSR shift register. During the EXTEST instruction, the output pins, as well as those bidirectional pins defined as outputs, are driven with their corresponding values from the BSR output register.

**Update-IR**

In this state, the IR output register is loaded from the IR shift register, and the current instruction is defined by the IR output register.

The following states have no effect on the normal or test operation of the processor other than as shown in Figure 82 on page 279:

- **Run-Test/Idle**—This state is an idle state between scan operations.
- **Select-DR-Scan**—This is the initial state of the test data register state transitions.
- **Select-IR-Scan**—This is the initial state of the Instruction Register state transitions.
- **Exit1-DR**—This state is entered to terminate the shifting process and enter the Update-DR state.
- **Exit1-IR**—This state is entered to terminate the shifting process and enter the Update-IR state.
- **Pause-DR**—This state is entered to temporarily stop the shifting process of a Test Data Register.
- **Pause-IR**—This state is entered to temporarily stop the shifting process of the Instruction Register.
- **Exit2-DR**—This state is entered in order to either terminate the shifting process and enter the Update-DR state or to resume shifting following the exit from the Pause-DR state.
- **Exit2-IR**—This state is entered in order to either terminate the shifting process and enter the Update-IR state or to resume shifting following the exit from the Pause-IR state.

# **11** *Test and Debug*

---

## **L1 Cache Inhibit**

---

### **Purpose**

The AMD-K6 3D processor provides a means for inhibiting the normal operation of its L1 instruction and data caches while still supporting an external Level-2 (L2) cache. This capability allows system designers to disable the L1 cache during the testing and debug of an L2 cache.

If the Cache Inhibit bit (bit 3) of Test Register 12 (TR12) is set to 0, the processor's L1 cache is enabled and operates as described in Chapter 8, "Cache Organization" on page 233. If the Cache Inhibit bit is set to 1, the L1 cache is disabled and no new cache lines are allocated. Even though new allocations do not occur, valid L1 cache lines remain valid and are read by the processor when a requested address hits a cache line. In addition, the processor continues to support inquire cycles initiated by the system logic, including the execution of writeback cycles when a modified cache line is hit.

While the L1 is inhibited, the processor continues to drive the PCD output signal appropriately, which system logic can use to control external L2 caching.

In order to completely disable the L1 cache so no valid lines exist in the cache, the Cache Inhibit bit must be set to 1 and the cache must be flushed in one of the following ways:

- By asserting the FLUSH# input signal
- By executing the WBINVD instruction
- By executing the INVD instruction (modified cache lines are not written back to memory)

## **Debug**

---

The processor implements the standard x86 debug functions, registers, and exceptions. In addition, the processor supports the I/O breakpoint debug extension. The debug feature assists programmers and system designers during software execution tracing by generating exceptions when one or more events occur during processor execution. The exception handler, or debugger, can be written to perform various tasks, such as displaying the conditions that caused the breakpoint to occur, displaying and modifying register or memory contents, or single-stepping through program execution.

The following sections describe the debug registers and the various types of breakpoints and exceptions that the processor supports.

### **Debug Registers**

Starting on page 284, Figures 83 through 86 show the 32-bit debug registers supported by the processor.



# 11 Test and Debug

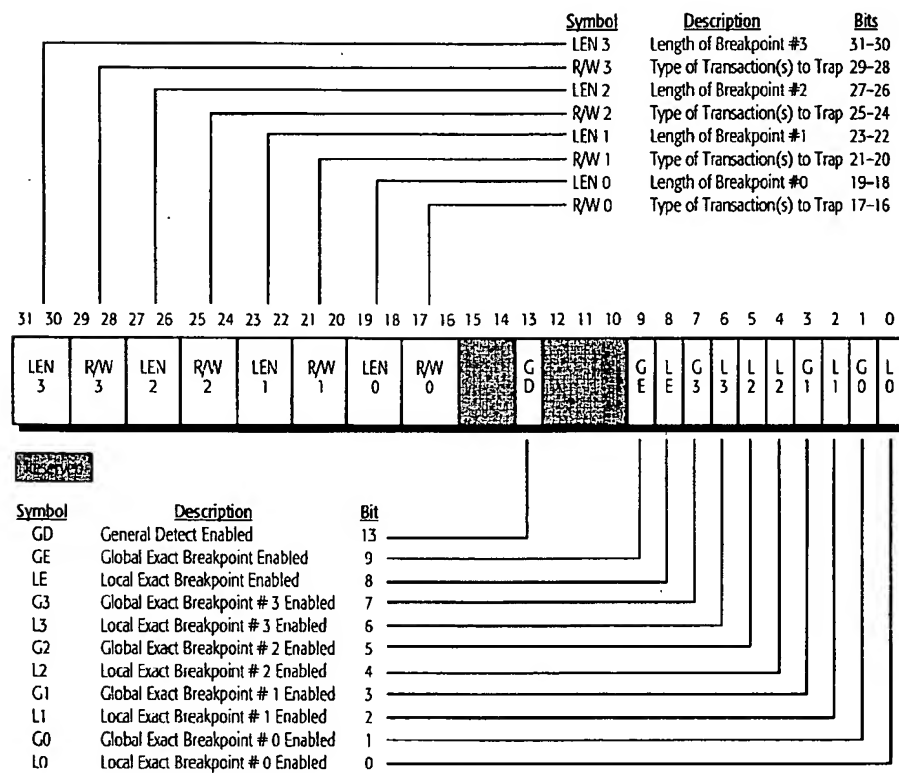


Figure 83. Debug Register DR7

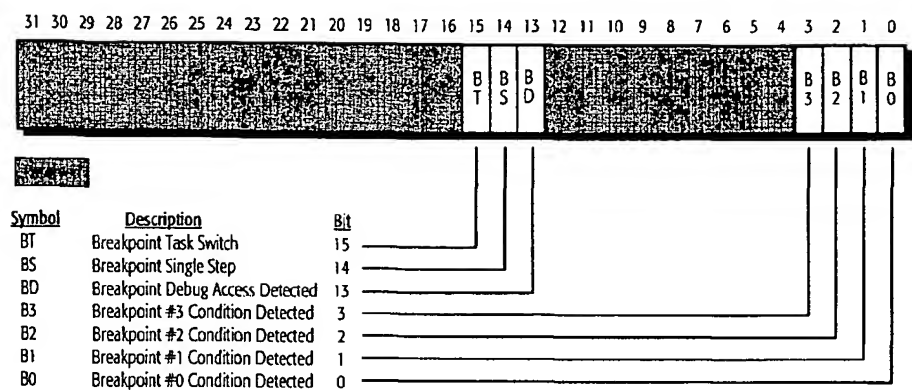


Figure 84. Debug Register DR6

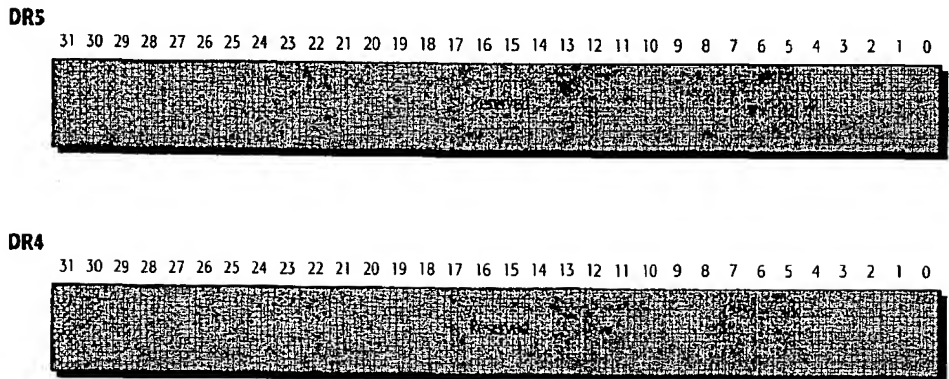
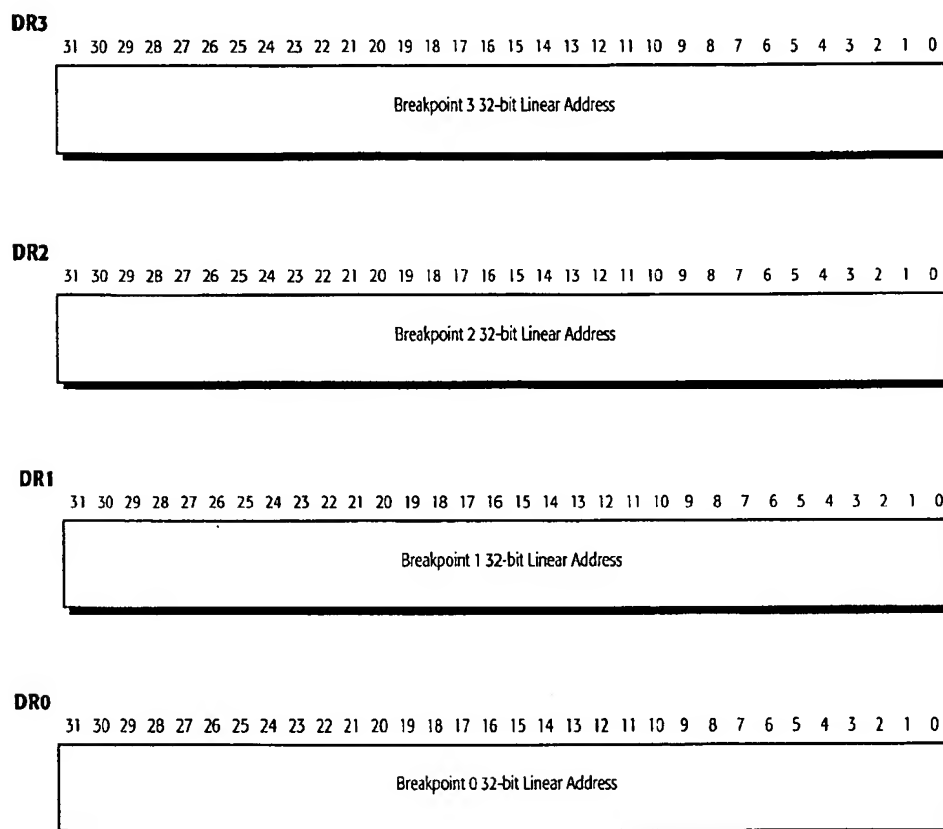


Figure 85. Debug Registers DR5 and DR4

# 11 Test and Debug



**Figure 86. Debug Registers DR3, DR2, DR1, and DR0**

**DR3–DR0** The processor allows the setting of up to four breakpoints. DR3–DR0 contain the linear addresses for breakpoint 3 through breakpoint 0, respectively, and are compared to the linear addresses of processor cycles to determine if a breakpoint occurs. Debug register DR7 defines the specific type of cycle that must occur in order for the breakpoint to occur.

**DR5–DR4** When debugging extensions are disabled (bit 3 of CR4 is set to 0), the DR5 and DR4 registers are mapped to DR7 and DR6, respectively, in order to be software compatible with previous generations of x86 processors. When debugging extensions are enabled (bit 3 of CR4 is set to 1), any attempt to load DR5 or

DR4 results in an undefined opcode exception. Likewise, any attempt to store DR5 or DR4 also results in an undefined opcode exception.

## **DR6**

If a breakpoint is enabled in DR7, and the breakpoint conditions as defined in DR7 occur, then the corresponding B-bit (B3–B0) in DR6 is set to 1. In addition, any other breakpoints defined using these particular breakpoint conditions are reported by the processor by setting the appropriate B-bits in DR6, regardless of whether these breakpoints are enabled or disabled. However, if a breakpoint is not enabled, a debug exception does not occur for that breakpoint.

If the processor decodes an instruction that writes or reads DR7 through DR0, the BD bit (bit 13) in DR6 is set to 1 (if enabled in DR7) and the processor generates a debug exception. This operation allows control to pass to the debugger prior to debug register access by software.

If the Trap Flag (bit 8) of the EFLAGS register is set to 1, the processor generates a debug exception after the successful execution of every instruction (single-step operation) and sets the BS bit (bit 14) in DR6 to indicate the source of the exception.

When the processor switches to a new task and the debug trap bit (T-bit) in the corresponding Task State Segment (TSS) is set to 1, the processor sets the BT bit (bit 15) in DR6 and generates a debug exception.

## **DR7**

When set to 1, L3–L0 locally enable breakpoints 3 through 0, respectively. L3–L0 are set to 0 whenever the processor executes a task switch. Setting L3–L0 to 0 disables the breakpoints and ensures that these particular debug exceptions are only generated for a specific task.

When set to 1, G3–G0 globally enable breakpoints 3 through 0, respectively. Unlike L3–L0, G3–G0 are not set to 0 whenever the processor executes a task switch. Not setting G3–G0 to 0 allows breakpoints to remain enabled across all tasks. If a breakpoint is enabled globally but disabled locally, the global enable overrides the local enable.

# 11 Test and Debug

The LE (bit 8) and GE (bit 9) bits in DR7 have no effect on the operation of the processor and are provided in order to be software compatible with previous generations of x86 processors.

When set to 1, the GD bit in DR7 (bit 13) enables the debug exception associated with the BD bit (bit 13) in DR6. This bit is set to 0 when a debug exception is generated.

LEN3–LEN0 and RW3–RW0 are two-bit fields in DR7 that specify the length and type of each breakpoint as defined in Table 66.

**Table 66. DR7 LEN and RW Definitions**

LEN Bits <sup>1</sup>	RW Bits	Breakpoint
00b	00b <sup>2</sup>	Instruction Execution
00b	01b	One-byte Data Write
01b		Two-byte Data Write
11b		Four-byte Data Write
00b	10b <sup>3</sup>	One-byte I/O Read or Write
01b		Two-byte I/O Read or Write
11b		Four-byte I/O Read or Write
00b	11b	One-byte Data Read or Write
01b		Two-byte Data Read or Write
11b		Four-byte Data Read or Write

**Notes:**

1. LEN bits equal to 10b is undefined.
2. When RW equals 00b, LEN must be equal to 00b.
3. When RW equals 10b, debugging extensions (DE) must be enabled (bit 3 of CR4 must be set to 1). If DE is set to 0, then RW equal to 10b is undefined.

## Debug Exceptions

A debug exception is categorized as either a debug trap or a debug fault. A debug trap calls the debugger following the execution of the instruction that caused the trap. A debug fault calls the debugger prior to the execution of the instruction that caused the fault. All debug traps and faults generate either an Interrupt 01h or an Interrupt 03h exception.

## **Interrupt 01h**

The following events are considered debug traps that cause the processor to generate an Interrupt 01h exception:

- Enabled breakpoints for data and I/O cycles
- Single Step Trap
- Task Switch Trap

The following events are considered debug faults that cause the processor to generate an Interrupt 01h exception:

- Enabled breakpoints for instruction execution
- BD bit in DR6 set to 1

## **Interrupt 03h**

The INT 3 instruction is defined in the x86 architecture as a breakpoint instruction. This instruction causes the processor to generate an Interrupt 03h exception. This exception is a debug trap because the debugger is called following the execution of the INT 3 instruction.

The INT 3 instruction is a one-byte instruction (opcode CCh) typically used to insert a breakpoint in software by writing CCh to the address of the first byte of the instruction to be trapped (the target instruction). Following the trap, if the target instruction is to be executed, the debugger must replace the INT 3 instruction with the first byte of the target instruction.

# **11** *Test and Debug*

---

**290**

---

## Clock Control

The AMD-K6 3D processor supports five modes of clock control. The processor can transition between these modes to maximize performance, to minimize power dissipation, or to provide a balance between performance and power. (See “Power Dissipation” on page 307 for the maximum power dissipation of the processor within the normal and reduced-power states.)

The five clock-control states supported are as follows:

- **Normal State:** The processor is running in Real Mode, Virtual-8086 Mode, Protected Mode, or System Management Mode (SMM). In this state, all clocks are running—including the external bus clock CLK and the internal processor clock—and the full features and functions of the processor are available.
- **Halt State:** This low-power state is entered following the successful execution of the HLT instruction. During this state, the internal processor clock is stopped.
- **Stop Grant State:** This low-power state is entered following the recognition of the assertion of the STPCLK# signal. During this state, the internal processor clock is stopped.
- **Stop Grant Inquire State:** This state is entered from the Halt state and the Stop Grant state as the result of a system-initiated inquire cycle.
- **Stop Clock State:** This low-power state is entered from the Stop Grant state when the CLK signal is stopped.



# 12 Clock Control

---

The following sections describe each of the four low-power states. Figure 87 on page 297 illustrates the clock control state transitions.

## Halt State

---

### Enter Halt State

During the execution of the HLT instruction, the processor executes a Halt special cycle. After BRDY# is sampled asserted during this cycle, and then EWBE# is also sampled asserted, the processor enters the Halt state in which the processor disables most of its internal clock distribution. In order to support the following operations, the internal phase-lock loop (PLL) still runs, and some internal resources are still clocked in the Halt state:

- **Inquire Cycles:** The processor continues to sample AHOLD, BOFF#, and HOLD in order to support inquire cycles that are initiated by the system logic. The processor transitions to the Stop Grant Inquire state during the inquire cycle. After returning to the Halt state following the inquire cycle, the processor does not execute another Halt special cycle.
- **Flush Cycles:** The processor continues to sample FLUSH#. If FLUSH# is sampled asserted, the processor performs the flush operation in the same manner as it is performed in the Normal state. Upon completing the flush operation, the processor executes the Halt special cycle which indicates the processor is in the Halt state.
- **Time Stamp Counter (TSC):** The TSC continues to count in the Halt state.
- **Signal Sampling:** The processor continues to sample INIT, INTR, NMI, RESET, and SMI#.

After entering the Halt state, all signals driven by the processor retain their state as they existed following the completion of the Halt special cycle.

## Exit Halt State

The processor remains in the Halt state until it samples INIT, INTR (if interrupts are enabled), NMI, RESET, or SMI# asserted. If any of these signals is sampled asserted, the processor returns to the Normal state and performs the corresponding operation. All of the normal requirements for recognition of these input signals apply within the Halt state.

## Stop Grant State

---

### Enter Stop Grant State

After recognizing the assertion of STPCLK#, the processor flushes its instruction pipelines, completes all pending and in-progress bus cycles, and acknowledges the STPCLK# assertion by executing a Stop Grant special bus cycle. After BRDY# is sampled asserted during this cycle, and then EWBE# is also sampled asserted, the processor enters the Stop Grant state. The Stop Grant state is like the Halt state in that the processor disables most of its internal clock distribution in the Stop Grant state. In order to support the following operations, the internal PLL still runs, and some internal resources are still clocked in the Stop Grant state:

- Inquire cycles: The processor transitions to the Stop Grant Inquire state during an inquire cycle. After returning to the Stop Grant state following the inquire cycle, the processor does not execute another Stop Grant special cycle.
- Time Stamp Counter (TSC): The TSC continues to count in the Stop Grant state.
- Signal Sampling: The processor continues to sample INIT, INTR, NMI, RESET, and SMI#.

FLUSH# is not recognized in the Stop Grant state (unlike while in the Halt state).

Upon entering the Stop Grant state, all signals driven by the processor retain their state as they existed following the completion of the Stop Grant special cycle.

## **12** *Clock Control*

---

### **Exit Stop Grant State**

The processor remains in the Stop Grant state until it samples STPCLK# negated or RESET asserted. If STPCLK# is sampled negated, the processor returns to the Normal state in less than 10 bus clock (CLK) periods. After the transition to the Normal state, the processor resumes execution at the instruction boundary on which STPCLK# was initially recognized.

If STPCLK# is recognized as negated in the Stop Grant state and subsequently sampled asserted prior to returning to the Normal state, the processor guarantees that a minimum of one instruction is executed prior to re-entering the Stop Grant state.

If INIT, INTR (if interrupts are enabled), FLUSH#, NMI, or SMI# are sampled asserted in the Stop Grant state, the processor latches the edge-sensitive signals (INIT, FLUSH#, NMI, and SMI#), but otherwise does not exit the Stop Grant state to service the interrupt. When the processor returns to the Normal state due to sampling STPCLK# negated, any pending interrupts are recognized after returning to the Normal state. To ensure their recognition, all of the normal requirements for these input signals apply within the Stop Grant state.

If RESET is sampled asserted in the Stop Grant state, the processor immediately returns to the Normal state and the reset process begins.

## **Stop Grant Inquire State**

---

### **Enter Stop Grant Inquire State**

The Stop Grant Inquire state is entered from the Stop Grant state or the Halt state when EADS# is sampled asserted during an inquire cycle initiated by the system logic. The processor responds to an inquire cycle in the same manner as in the Normal state by driving HIT# and HITM#. If the inquire cycle hits a modified data cache line, the processor performs a writeback cycle.

### **Exit Stop Grant Inquire State**

Following the completion of any writeback, the processor returns to the state from which it entered the Stop Grant Inquire state.

## **Stop Clock State**

---

### **Enter Stop Clock State**

If the CLK signal is stopped while the processor is in the Stop Grant state, the processor enters the Stop Clock state. Because all internal clocks and the PLL are not running in the Stop Clock state, the Stop Clock state represents the minimum-power state of all clock control states. The CLK signal must be held Low while it is stopped.

The Stop Clock state cannot be entered from the Halt state.

INTR is the only input signal that is allowed to change states while the processor is in the Stop Clock state. However, INTR is not sampled until the processor returns to the Stop Grant state. All other input signals must remain unchanged in the Stop Clock state.

---

## **12** *Clock Control*

---

### **Exit Stop Clock State**

The processor returns to the Stop Grant state from the Stop Clock state after the CLK signal is started and the internal PLL has stabilized. PLL stabilization is achieved after the CLK signal has been running within its specification for a minimum of 1.0 ms.

The frequency of CLK when exiting the Stop Clock state can be different than the frequency of CLK when entering the Stop Clock state.

The state of the BF[2:0] signals when exiting the Stop Clock state is ignored because the BF[2:0] signals are only sampled during the falling transition of RESET.

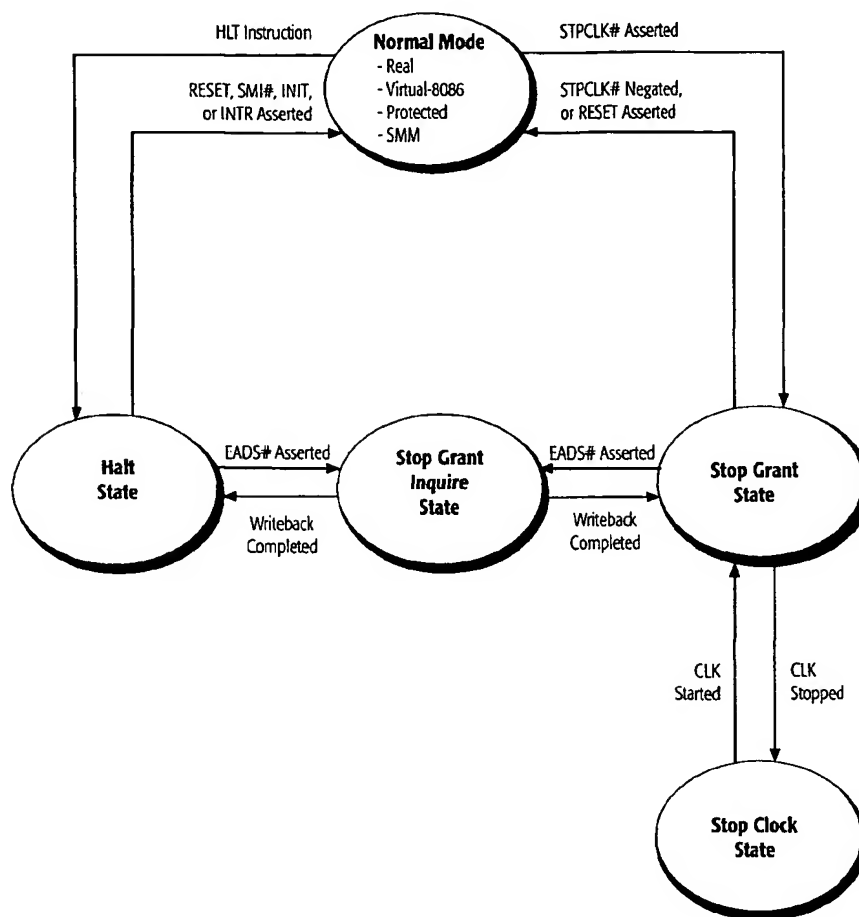


Figure 87. Clock Control State Transitions

---

## **12** *Clock Control*

---

# 13

## Power and Grounding

### Power Connections

---

The AMD-K6 3D processor is a dual voltage device. Two separate supply voltages are required— $V_{CC2}$  and  $V_{CC3}$ .  $V_{CC2}$  provides the core voltage for the processor and  $V_{CC3}$  provides the I/O voltage. See Chapter 14, “Electrical Data” on page 303 for the value and range of  $V_{CC2}$  and  $V_{CC3}$ .

There are 28  $V_{CC2}$ , 32  $V_{CC3}$ , and 68  $V_{SS}$  pins on the processor. (See “Pin Designations” on page 342 for all power and ground pin designations.) The large number of power and ground pins are provided to ensure that the processor and package maintain a clean and stable power distribution network.

For proper operation and functionality, all  $V_{CC2}$ ,  $V_{CC3}$ , and  $V_{SS}$  pins must be connected to the appropriate planes in the circuit board. The power planes have been arranged in a pattern to simplify routing and minimize crosstalk on the circuit board. The isolation region between two voltage planes must be at least 0.254mm if they are in the same layer of the circuit board. (See Figure 88 on page 300.) In order to maintain a low-impedance current sink and reference, the ground plane must never be split.



# 13 Power and Grounding

Although the processor has two separate supply voltages, there are no special power sequencing requirements. The best procedure is to minimize the time between which  $V_{CC2}$  and  $V_{CC3}$  are either both on or both off.

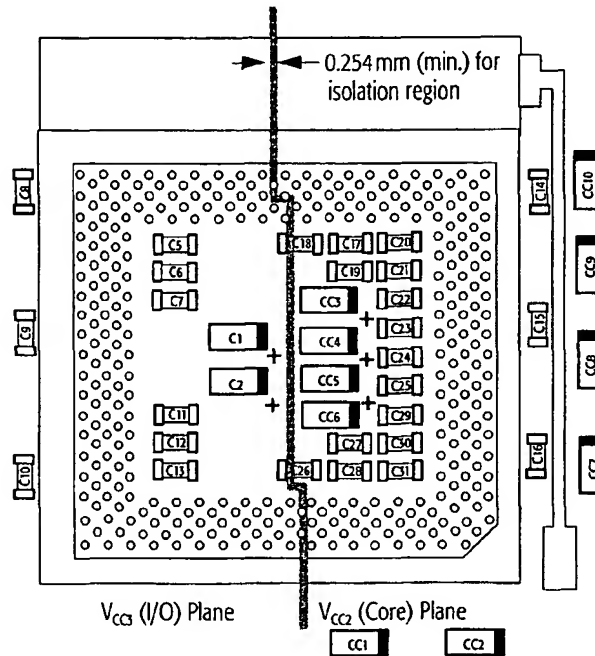


Figure 88. Suggested Component Placement

## **Decoupling Recommendations**

---

In addition to the isolation region mentioned in “Power Connections” on page 299, adequate decoupling capacitance is required between the two system power planes and the ground plane to minimize ringing and to provide a low-impedance path for return currents. Suggested decoupling capacitor placement is shown in Figure 88 on page 300.

Surface mounted capacitors should be used under the processor’s ZIF socket to minimize resistance and inductance in the lead lengths while maintaining minimal height. For information and recommendations about the specific value, quantity, and location of capacitors, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).

## **Pin Connection Requirements**

---

For proper operation, the following requirements for signal pin connections must be met:

- Do not drive address and data signals into large capacitive loads at high frequencies. If necessary, use buffer chips to drive large capacitive loads.
- Leave all NC (no-connect) pins unconnected.
- Unused inputs should always be connected to an appropriate signal level.
  - Active Low inputs that are not being used should be connected to  $V_{CC3}$  through a 20-kohm pullup resistor.
  - Active High inputs that are not being used should be connected to GND through a pulldown resistor.
- Reserved signals can be treated in one of the following ways:
  - As no-connect (NC) pins, in which case these pins are left unconnected
  - As pins connected to the system logic as defined by the industry-standard Pentium interface (Socket 7)
  - Any combination of NC and Socket 7 pins
- Keep trace lengths to a minimum.

---

# **13** *Power and Grounding*

---

**302**

---

## Electrical Data

### Introduction

---

This chapter contains electrical data that is subject to change. For the latest values, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).

### Operating Ranges

---

The functional operation of the AMD-K6 3D processor is guaranteed if the voltage and temperature parameters are within the limits defined in Table 67.

**Table 67. Operating Ranges**

Parameter	Minimum	Typical	Maximum	Comments
$V_{CC2}$	2.1 V	2.2 V	2.3 V	Note
$V_{CC3}$	3.135 V	3.30 V	3.6 V	Note
$T_{CASE}$	0°C		70°C	
<b>Note:</b> $V_{CC2}$ and $V_{CC3}$ are referenced from $V_{SS}$				

# 14 Electrical Data

## Absolute Ratings

Functional operation of the processor is not guaranteed beyond the operating ranges listed in Table 67 on page 303. Exposure to conditions outside these operating ranges for extended periods of time can affect long-term reliability. Permanent damage can occur if the absolute ratings listed in Table 68 are exceeded.

**Table 68. Absolute Ratings**

Parameter	Minimum	Maximum	Comments
$V_{CC2}$	-0.5 V	2.5 V	
$V_{CC3}$	-0.5 V	3.6 V	
$V_{PIN}$	-0.5 V	$V_{CC3} + 0.5$ V and $\leq 4.0$ V	Note
$T_{CASE}$ (under bias)	-65°C	+110°C	
$T_{STORAGE}$	-65°C	+150°C	
<b>Note:</b> $V_{PIN}$ (the voltage on any I/O pin) must not be greater than 0.5 V above the voltage being applied to $V_{CC3}$ . In addition, the $V_{PIN}$ voltage must never exceed 4.0 V.			

## DC Characteristics

The DC characteristics of the processor are shown in Table 69.

Table 69. DC Characteristics

Symbol	Parameter Description	Preliminary Data		Comments
		Min	Max	
$V_{IL}$	Input Low Voltage	-0.3 V	+0.8 V	
$V_{IH}$	Input High Voltage	2.0 V	$V_{CC3} + 0.3$ V	Note 1
$V_{OL}$	Output Low Voltage		0.4 V	$I_{OL} = 4.0$ -mA load
$V_{OH}$	Output High Voltage	2.4 V		$I_{OH} = 3.0$ -mA load
$I_{CC2}$	2.2 V Power Supply Current		6.50 A	233 MHz, Note 2,7
			6.90 A	250 MHz, Note 2,8
			7.35 A	266 MHz, Note 2,7
			8.45 A	300 MHz, Note 2,9
			9.40 A	333 MHz, Note 2,7
$I_{CC3}$	3.3 V Power Supply Current		0.52 A	233 MHz, Note 3,7
			0.53 A	250 MHz, Note 3,8
			0.54 A	266 MHz, Note 3,7
			0.56 A	300 MHz, Note 3,9
			0.58 A	333 MHz, Note 3,7
$I_{LI}$	Input Leakage Current		$\pm 15$ $\mu$ A	Note 4
$I_{LO}$	Output Leakage Current		$\pm 15$ $\mu$ A	Note 4
$I_{IL}$	Input Leakage Current Bias with Pullup		-400 $\mu$ A	Note 5
$I_{IH}$	Input Leakage Current Bias with Pulldown		200 $\mu$ A	Note 6
$C_{IN}$	Input Capacitance		10 pF	
$C_{OUT}$	Output Capacitance		15 pF	
<b>Notes:</b> <ol style="list-style-type: none"> <li>1. <math>V_{CC3}</math> refers to the voltage being applied to <math>V_{CC3}</math> during functional operation.</li> <li>2. <math>V_{CC2} = 2.3</math> V – The maximum power supply current must be taken into account when designing a power supply.</li> <li>3. <math>V_{CC3} = 3.6</math> V – The maximum power supply current must be taken into account when designing a power supply.</li> <li>4. Refers to inputs and I/O without an internal pullup resistor and <math>0 \leq V_{IN} \leq V_{CC3}</math>.</li> <li>5. Refers to inputs with an internal pullup and <math>V_{IL} = 0.4</math> V.</li> <li>6. Refers to inputs with an internal pulldown and <math>V_{IH} = 2.4</math> V.</li> <li>7. CLK frequency equals 66 MHz.</li> <li>8. CLK frequency equals 100 MHz.</li> <li>9. This specification applies to components using a CLK frequency of 66 MHz or 100 MHz.</li> </ol>				

# 14 Electrical Data

Table 69. DC Characteristics (continued)

Symbol	Parameter Description	Preliminary Data		Comments
		Min	Max	
$C_{OUT}$	I/O Capacitance		20 pF	
$C_{CLK}$	CLK Capacitance		10 pF	
$C_{TIN}$	Test Input Capacitance (TDI, TMS, TRST#)		10 pF	
$C_{TOUT}$	Test Output Capacitance (TDO)		15 pF	
$C_{TCK}$	TCK Capacitance		10 pF	
<b>Notes:</b> <ol style="list-style-type: none"> <li>1. <math>V_{CC3}</math> refers to the voltage being applied to <math>V_{CC3}</math> during functional operation.</li> <li>2. <math>V_{CC2} = 2.3\text{ V}</math> – The maximum power supply current must be taken into account when designing a power supply.</li> <li>3. <math>V_{CC3} = 3.6\text{ V}</math> – The maximum power supply current must be taken into account when designing a power supply.</li> <li>4. Refers to inputs and I/O without an internal pullup resistor and <math>0 \leq V_{IH} \leq V_{CC3}</math>.</li> <li>5. Refers to inputs with an internal pullup and <math>V_{IL} = 0.4\text{ V}</math>.</li> <li>6. Refers to inputs with an internal pulldown and <math>V_{IH} = 2.4\text{ V}</math>.</li> <li>7. CLK frequency equals 66 MHz.</li> <li>8. CLK frequency equals 100 MHz.</li> <li>9. This specification applies to components using a CLK frequency of 66 MHz or 100 MHz.</li> </ol>				

## Power Dissipation

Table 70 contains the typical and maximum power dissipation of the processor during normal and reduced power states.

**Table 70. Typical and Maximum Power Dissipation**

Clock Control State	233 MHz <sup>6</sup>	250 MHz <sup>7</sup>	266 MHz <sup>6</sup>	300 MHz <sup>8</sup>	333 MHz <sup>6</sup>	Comments
Normal (Maximum Thermal Power)	13.50 W	13.85 W	14.70 W	17.20 W	19.00 W	Note 1, 2
Normal (Typical Thermal Power)	8.10 W	8.30 W	8.85 W	10.35 W	11.40 W	Note 3
Stop Grant / Halt (Maximum)	2.46 W	2.47 W	2.48 W	2.50 W	2.52 W	Note 4
Stop Clock (Maximum)	2.25 W	2.25 W	2.25 W	2.25 W	2.25 W	Note 5
<b>Notes:</b> <ol style="list-style-type: none"> <li>1. The maximum power dissipated in the normal clock control state must be taken into account when designing a solution for thermal dissipation for the AMD-K6 3D processor.</li> <li>2. Maximum power is determined for the worst-case instruction sequence or function for the listed clock control states with <math>V_{CC2} = 2.2</math> V and <math>V_{CC3} = 3.3</math> V.</li> <li>3. Typical power is determined for the typical instruction sequences or functions associated with normal system operation with <math>V_{CC2} = 2.2</math> V and <math>V_{CC3} = 3.3</math> V.</li> <li>4. The CLK signal and the internal PLL are still running but most internal clocking has stopped.</li> <li>5. The CLK signal, the internal PLL, and all internal clocking has stopped.</li> <li>6. CLK frequency equals 66 MHz.</li> <li>7. CLK frequency equals 100 MHz.</li> <li>8. This specification applies to components using a CLK frequency of 66 MHz or 100 MHz.</li> </ol>						



---

## **14** *Electrical Data*

---

**308**

## I/O Buffer Characteristics

### Introduction

---

This chapter contains data that is subject to change. For the latest I/O buffer characteristics, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).

All of the AMD-K6 3D processor inputs, outputs, and bidirectional buffers are implemented using a 3.3V buffer design. In addition, a subset of the processor I/O buffers include a second, higher drive strength option. These buffers can be configured to provide the higher drive strength for applications that place a heavier load on these I/O signals.

AMD has developed two I/O buffer models that represent the characteristics of each of the two possible drive strength configurations supported by the processor. These two models are called the Standard I/O Model and the Strong I/O Model.

AMD developed the two models to allow system designers to perform analog simulations of processor signals that interface with the system logic. Analog simulations are used to determine a signal's time of flight from source to destination and to ensure that the system's signal quality requirements are met. Signal quality measurements include overshoot, undershoot, slope reversal, and ringing.

# 15 I/O Buffer Characteristics

## Selectable Drive Strength

The processor samples the BRDYC# input during the falling transition of RESET to configure the drive strength of A[20:3], ADS#, HITM# and W/R#. If BRDYC# is 0 during the fall of RESET, these particular outputs are configured using the higher drive strength. If BRDYC# is 1 during the fall of RESET, the standard drive strength is selected for all I/O buffers.

Table 71 shows the relationship between BRDYC# and the two available drive strengths — K6STD and K6STG.

Table 71. A[20:3], ADS#, HITM#, and W/R# Strength Selection

Drive Strength	BRDYC#	I/O Buffer Name
Strength 1 (standard)	1	K6STD
Strength 2 (strong)	0	K6STG

## I/O Buffer Model

AMD provides models of the processor I/O buffers for system designers to use in board-level simulations. These I/O buffer models conform to the *I/O Buffer Information Specification (IBIS), Version 2.1*. The Standard I/O Model uses K6STD, the standard I/O buffer representation, for all I/O buffers. The Strong I/O Model uses K6STG, the stronger I/O buffer representation for A[20:3], ADS#, HITM#, and W/R#, and uses K6STD for the remainder of the I/O buffers.

Both I/O models contain voltage versus current (V/I) and voltage versus time (V/T) data tables for accurate modeling of I/O buffer behavior.

The following list characterizes the properties of each I/O buffer model:

- All data tables contain minimum, typical, and maximum values to allow for worst-case, typical, and best-case simulations, respectively.
- The pullup, pulldown, power clamp, and ground clamp device V/I tables contain enough data points to accurately represent the nonlinear nature of the V/I curves. In addition, the voltage ranges provided in these tables extend beyond

## I/O Buffer Characteristics **15**

the normal operating range of the processor for those simulators that yield more accurate results based on this wider range. Figure 89 and Figure 90 illustrate the min/typ/max pulldown and pullup V/I curves for K6STD between 0V and 3.3V.

- The rising and falling ramp rates are specified.
- The min/typ/max  $V_{CC3}$  operating range is specified as 3.135V, 3.3V, and 3.6V, respectively.
- $V_{il} = 0.8V$ ,  $V_{ih} = 2.0V$ , and  $V_{meas} = 1.5V$
- The R/L/C of the package is modeled.
- The capacitance of the silicon die is modeled.
- The model assumes the test load is 0 capacitance, resistance, inductance, and voltage.

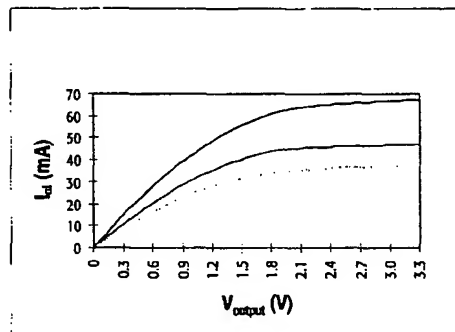


Figure 89. K6STD Pulldown V/I Curves

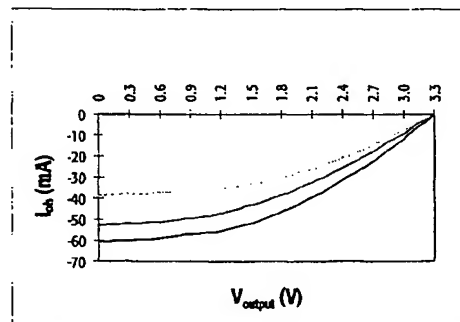


Figure 90. K6STD Pullup V/I Curves

---

# **15** *I/O Buffer Characteristics*

---

## **I/O Model Application Note**

---

For the AMD-K6 3D processor I/O Buffer IBIS Models and their application, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).

## **I/O Buffer AC and DC Characteristics**

---

See Chapter 16, “Signal Switching Characteristics” on page 313 for the processor AC timing specifications.

See Chapter 14, “Electrical Data” on page 303 for the processor DC specifications.

# 16

## Signal Switching Characteristics

### Introduction

---

This chapter contains data that is subject to change. For the latest signal switching characteristics, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).

The AMD-K6 3D processor signal switching characteristics are presented in Table 72 through Table 81 starting on page 314. Valid delay, float, setup, and hold timing specifications are listed. These specifications are provided for the system designer to determine if the timings necessary for the processor to interface with the system logic are met. Table 72 and Table 73 contain the switching characteristics of the CLK input. Table 74 through Table 77 contain the timings for the normal operation signals. Table 78 and Table 79 contain the timings for RESET and the configuration signals. Table 80 and Table 81 contain the timings for the test operation signals.

All signal timings provided are:

- Measured between CLK, TCK, or RESET at 1.5 V and the corresponding signal at 1.5 V—this applies to input and output signals that are switching from Low to High, or from High to Low
- Based on input signals applied at a slew rate of 1 V/ns between 0 V and 3 V (rising) and 3 V to 0 V (falling)

**313**

---

# 16 Signal Switching Characteristics

- Valid within the operating ranges given in “Operating Ranges” on page 303
- Based on a load capacitance ( $C_L$ ) of 0 pF

## CLK Switching Characteristics

Table 72 and Table 73 contain the switching characteristics of the CLK input to the processor for 100-MHz and 66-MHz bus operation, respectively, as measured at the voltage levels indicated by Figure 91 on page 315.

The CLK Period Stability specifies the variance (jitter) allowed between successive periods of the CLK input measured at 1.5 V. This parameter must be considered as one of the elements of clock skew between the processor and the system logic.

## Clock Switching Characteristics for 100-MHz Bus Operation

Table 72. CLK Switching Characteristics for 100-MHz Bus Operation

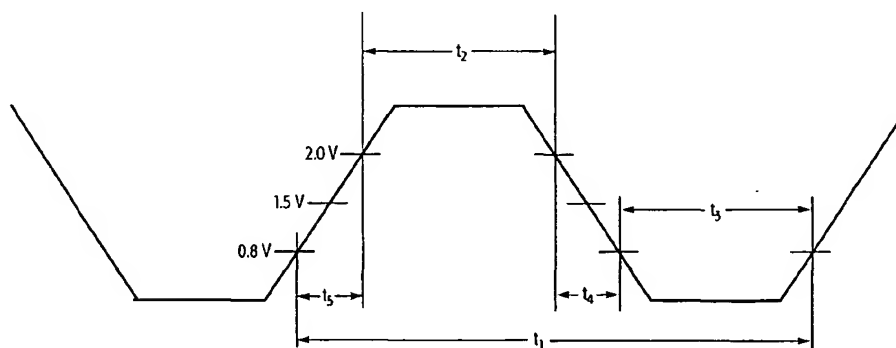
Symbol	Parameter Description	Advance Data		Figure	Comments
		Min	Max		
	Frequency		100 MHz		In Normal Mode
$t_1$	CLK Period	10.0 ns		91	In Normal Mode
$t_2$	CLK High Time	3.0 ns		91	
$t_3$	CLK Low Time	3.0 ns		91	
$t_4$	CLK Fall Time	0.15 ns	1.5 ns	91	
$t_5$	CLK Rise Time	0.15 ns	1.5 ns	91	
	CLK Period Stability		$\pm 250$ ps		Note
<b>Note:</b> <i>Jitter frequency power spectrum peaking must occur at frequencies greater than (Frequency of CLK)/3 or less than 500 kHz.</i>					

## Clock Switching Characteristics for 66-MHz Bus Operation

**Table 73. CLK Switching Characteristics for 66-MHz Bus Operation**

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
	Frequency	33.3 MHz	66.6 MHz		In Normal Mode
$t_1$	CLK Period	15.0 ns	30.0 ns	91	In Normal Mode
$t_2$	CLK High Time	4.0 ns		91	
$t_3$	CLK Low Time	4.0 ns		91	
$t_4$	CLK Fall Time	0.15 ns	1.5 ns	91	
$t_5$	CLK Rise Time	0.15 ns	1.5 ns	91	
	CLK Period Stability		$\pm 250$ ps		Note

**Note:**  
*Jitter frequency power spectrum peaking must occur at frequencies greater than (Frequency of CLK)/3 or less than 500 KHz.*



**Figure 91. CLK Waveform**



# 16 Signal Switching Characteristics

## Valid Delay, Float, Setup, and Hold Timings

Valid delay and float timings are given for output signals during functional operation and are given relative to the rising edge of CLK. During boundary-scan testing, valid delay and float timings for output signals are with respect to the falling edge of TCK. The maximum valid delay timings are provided to allow a system designer to determine if setup times to the system logic can be met. Likewise, the minimum valid delay timings are used to analyze hold times to the system logic.

The setup and hold time requirements for the processor input signals must be met by the system logic to assure the proper operation of the processor. The setup and hold timings during functional and boundary-scan test mode are given relative to the rising edge of CLK and TCK, respectively.

## Output Delay Timings for 100-MHz Bus Operation

Table 74. Output Delay Timings for 100-MHz Bus Operation

Symbol	Parameter Description	Advance Data		Figure	Comments
		Min	Max		
$t_6$	A[31:3] Valid Delay	1.1 ns	4.0 ns	93	
$t_7$	A[31:3] Float Delay		7.0 ns	94	
$t_8$	ADS# Valid Delay	1.0 ns	4.0 ns	93	
$t_9$	ADS# Float Delay		7.0 ns	94	
$t_{10}$	ADSC# Valid Delay	1.0 ns	4.0 ns	93	
$t_{11}$	ADSC# Float Delay		7.0 ns	94	
$t_{12}$	AP Valid Delay	1.0 ns	5.5 ns	93	
$t_{13}$	AP Float Delay		7.0 ns	94	
$t_{14}$	APCHK# Valid Delay	1.0 ns	4.5 ns	93	
$t_{15}$	BE[7:0]# Valid Delay	1.0 ns	4.0 ns	93	
$t_{16}$	BE[7:0]# Float Delay		7.0 ns	94	
$t_{17}$	BREQ Valid Delay	1.0 ns	4.0 ns	93	
$t_{18}$	CACHE# Valid Delay	1.0 ns	4.0 ns	93	

# Signal Switching Characteristics **16**

**Table 74. Output Delay Timings for 100-MHz Bus Operation (continued)**

Symbol	Parameter Description	Advance Data		Figure	Comments
		Min	Max		
t <sub>19</sub>	CACHE# Float Delay		7.0 ns	94	
t <sub>20</sub>	D/C# Valid Delay	1.0 ns	4.0 ns	93	
t <sub>21</sub>	D/C# Float Delay		7.0 ns	94	
t <sub>22</sub>	D[63:0] Write Data Valid Delay	1.3 ns	4.5 ns	93	
t <sub>23</sub>	D[63:0] Write Data Float Delay		7.0 ns	94	
t <sub>24</sub>	DP[7:0] Write Data Valid Delay	1.3 ns	4.5 ns	93	
t <sub>25</sub>	DP[7:0] Write Data Float Delay		7.0 ns	94	
t <sub>26</sub>	FERR# Valid Delay	1.0 ns	4.5 ns	93	
t <sub>27</sub>	HIT# Valid Delay	1.0 ns	4.0 ns	93	
t <sub>28</sub>	HITM# Valid Delay	1.1 ns	4.0 ns	93	
t <sub>29</sub>	HLDA Valid Delay	1.0 ns	4.0 ns	93	
t <sub>30</sub>	LOCK# Valid Delay	1.1 ns	4.0 ns	93	
t <sub>31</sub>	LOCK# Float Delay		7.0 ns	94	
t <sub>32</sub>	M/IO# Valid Delay	1.0 ns	4.0 ns	93	
t <sub>33</sub>	M/IO# Float Delay		7.0 ns	94	
t <sub>34</sub>	PCD Valid Delay	1.0 ns	4.0 ns	93	
t <sub>35</sub>	PCD Float Delay		7.0 ns	94	
t <sub>36</sub>	PCHK# Valid Delay	1.0 ns	4.5 ns	93	
t <sub>37</sub>	PWT Valid Delay	1.0 ns	4.0 ns	93	
t <sub>38</sub>	PWT Float Delay		7.0 ns	94	
t <sub>39</sub>	SCYC Valid Delay	1.0 ns	4.0 ns	93	
t <sub>40</sub>	SCYC Float Delay		7.0 ns	94	
t <sub>41</sub>	SMIACK# Valid Delay	1.0 ns	4.0 ns	93	
t <sub>42</sub>	W/R# Valid Delay	1.0 ns	4.0 ns	93	
t <sub>43</sub>	W/R# Float Delay		7.0 ns	94	

# 16 Signal Switching Characteristics

## Input Setup and Hold Timings for 100-MHz Bus Operation

Table 75. Input Setup and Hold Timings for 100-MHz Bus Operation

Symbol	Parameter Description	Advance Data		Figure	Comments
		Min	Max		
t <sub>44</sub>	A[31:5] Setup Time	3.0 ns		95	
t <sub>45</sub>	A[31:5] Hold Time	1.0 ns		95	
t <sub>46</sub>	A20M# Setup Time	3.0 ns		95	Note 1
t <sub>47</sub>	A20M# Hold Time	1.0 ns		95	Note 1
t <sub>48</sub>	AHOLD Setup Time	3.5 ns		95	
t <sub>49</sub>	AHOLD Hold Time	1.0 ns		95	
t <sub>50</sub>	AP Setup Time	1.7 ns		95	
t <sub>51</sub>	AP Hold Time	1.0 ns		95	
t <sub>52</sub>	BOFF# Setup Time	3.5 ns		95	
t <sub>53</sub>	BOFF# Hold Time	1.0 ns		95	
t <sub>54</sub>	BRDY# Setup Time	3.0 ns		95	
t <sub>55</sub>	BRDY# Hold Time	1.0 ns		95	
t <sub>56</sub>	BRDYC# Setup Time	3.0 ns		95	
t <sub>57</sub>	BRDYC# Hold Time	1.0 ns		95	
t <sub>58</sub>	D[63:0] Read Data Setup Time	1.7 ns		95	
t <sub>59</sub>	D[63:0] Read Data Hold Time	1.5 ns		95	
t <sub>60</sub>	DP[7:0] Read Data Setup Time	1.7 ns		95	
t <sub>61</sub>	DP[7:0] Read Data Hold Time	1.5 ns		95	
t <sub>62</sub>	EADS# Setup Time	3.0 ns		95	
t <sub>63</sub>	EADS# Hold Time	1.0 ns		95	
t <sub>64</sub>	EWBE# Setup Time	1.7 ns		95	
t <sub>65</sub>	EWBE# Hold Time	1.0 ns		95	
t <sub>66</sub>	FLUSH# Setup Time	1.7 ns		95	Note 2

**Notes:**

1. These level-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must be asserted for a minimum pulse width of two clocks.
2. These edge-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must have been negated at least two clocks prior to assertion and must remain asserted at least two clocks.

# Signal Switching Characteristics **16**

**Table 75. Input Setup and Hold Timings for 100-MHz Bus Operation (continued)**

Symbol	Parameter Description	Advance Data		Figure	Comments
		Min	Max		
t <sub>67</sub>	FLUSH# Hold Time	1.0 ns		95	Note 2
t <sub>68</sub>	HOLD Setup Time	1.7 ns		95	
t <sub>69</sub>	HOLD Hold Time	1.5 ns		95	
t <sub>70</sub>	IGNNE# Setup Time	1.7 ns		95	Note 1
t <sub>71</sub>	IGNNE# Hold Time	1.0 ns		95	Note 1
t <sub>72</sub>	INIT Setup Time	1.7 ns		95	Note 2
t <sub>73</sub>	INIT Hold Time	1.0 ns		95	Note 2
t <sub>74</sub>	INTR Setup Time	1.7 ns		95	Note 1
t <sub>75</sub>	INTR Hold Time	1.0 ns		95	Note 1
t <sub>76</sub>	INV Setup Time	1.7 ns		95	
t <sub>77</sub>	INV Hold Time	1.0 ns		95	
t <sub>78</sub>	KEN# Setup Time	3.0 ns		95	
t <sub>79</sub>	KEN# Hold Time	1.0 ns		95	
t <sub>80</sub>	NA# Setup Time	1.7 ns		95	
t <sub>81</sub>	NA# Hold Time	1.0 ns		95	
t <sub>82</sub>	NMI Setup Time	1.7 ns		95	Note 2
t <sub>83</sub>	NMI Hold Time	1.0 ns		95	Note 2
t <sub>84</sub>	SMI# Setup Time	1.7 ns		95	Note 2
t <sub>85</sub>	SMI# Hold Time	1.0 ns		95	Note 2
t <sub>86</sub>	STPCLK# Setup Time	1.7 ns		95	Note 1
t <sub>87</sub>	STPCLK# Hold Time	1.0 ns		95	Note 1
t <sub>88</sub>	WB/WT# Setup Time	1.7 ns		95	
t <sub>89</sub>	WB/WT# Hold Time	1.0 ns		95	

**Notes:**

1. These level-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must be asserted for a minimum pulse width of two clocks.
2. These edge-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must have been negated at least two clocks prior to assertion and must remain asserted at least two clocks.

# 16 Signal Switching Characteristics

## Output Delay Timings for 66-MHz Bus Operation

Table 76. Output Delay Timings for 66-MHz Bus Operation

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
t <sub>6</sub>	A[31:3] Valid Delay	1.1 ns	6.3 ns	93	
t <sub>7</sub>	A[31:3] Float Delay		10.0 ns	94	
t <sub>8</sub>	ADS# Valid Delay	1.0 ns	6.0 ns	93	
t <sub>9</sub>	ADS# Float Delay		10.0 ns	94	
t <sub>10</sub>	ADSC# Valid Delay	1.0 ns	7.0 ns	93	
t <sub>11</sub>	ADSC# Float Delay		10.0 ns	94	
t <sub>12</sub>	AP Valid Delay	1.0 ns	8.5 ns	93	
t <sub>13</sub>	AP Float Delay		10.0 ns	94	
t <sub>14</sub>	APCHK# Valid Delay	1.0 ns	8.3 ns	93	
t <sub>15</sub>	BE[7:0]# Valid Delay	1.0 ns	7.0 ns	93	
t <sub>16</sub>	BE[7:0]# Float Delay		10.0 ns	94	
t <sub>17</sub>	BREQ Valid Delay	1.0 ns	8.0 ns	93	
t <sub>18</sub>	CACHE# Valid Delay	1.0 ns	7.0 ns	93	
t <sub>19</sub>	CACHE# Float Delay		10.0 ns	94	
t <sub>20</sub>	D/C# Valid Delay	1.0 ns	7.0 ns	93	
t <sub>21</sub>	D/C# Float Delay		10.0 ns	94	
t <sub>22</sub>	D[63:0] Write Data Valid Delay	1.3 ns	7.5 ns	93	
t <sub>23</sub>	D[63:0] Write Data Float Delay		10.0 ns	94	
t <sub>24</sub>	DP[7:0] Write Data Valid Delay	1.3 ns	7.5 ns	93	
t <sub>25</sub>	DP[7:0] Write Data Float Delay		10.0 ns	94	
t <sub>26</sub>	FERR# Valid Delay	1.0 ns	8.3 ns	93	
t <sub>27</sub>	HIT# Valid Delay	1.0 ns	6.8 ns	93	
t <sub>28</sub>	HITM# Valid Delay	1.1 ns	6.0 ns	93	
t <sub>29</sub>	HLDA Valid Delay	1.0 ns	6.8 ns	93	
t <sub>30</sub>	LOCK# Valid Delay	1.1 ns	7.0 ns	93	
t <sub>31</sub>	LOCK# Float Delay		10.0 ns	94	
t <sub>32</sub>	M/IO# Valid Delay	1.0 ns	5.9 ns	93	

# Signal Switching Characteristics **16**

**Table 76. Output Delay Timings for 66-MHz Bus Operation (continued)**

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
t <sub>33</sub>	M/IO# Float Delay		10.0 ns	94	
t <sub>34</sub>	PCD Valid Delay	1.0 ns	7.0 ns	93	
t <sub>35</sub>	PCD Float Delay		10.0 ns	94	
t <sub>36</sub>	PCHK# Valid Delay	1.0 ns	7.0 ns	93	
t <sub>37</sub>	PWT Valid Delay	1.0 ns	7.0 ns	93	
t <sub>38</sub>	PWT Float Delay		10.0 ns	94	
t <sub>39</sub>	SCYC Valid Delay	1.0 ns	7.0 ns	93	
t <sub>40</sub>	SCYC Float Delay		10.0 ns	94	
t <sub>41</sub>	SMIACK# Valid Delay	1.0 ns	7.3 ns	93	
t <sub>42</sub>	W/R# Valid Delay	1.0 ns	7.0 ns	93	
t <sub>43</sub>	W/R# Float Delay		10.0 ns	94	

# 16 Signal Switching Characteristics

## Input Setup and Hold Timings for 66-MHz Bus Operation

Table 77. Input Setup and Hold Timings for 66-MHz Bus Operation

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
t <sub>44</sub>	A[31:5] Setup Time	6.0 ns		95	
t <sub>45</sub>	A[31:5] Hold Time	1.0 ns		95	
t <sub>46</sub>	A20M# Setup Time	5.0 ns		95	Note 1
t <sub>47</sub>	A20M# Hold Time	1.0 ns		95	Note 1
t <sub>48</sub>	AHOLD Setup Time	5.5 ns		95	
t <sub>49</sub>	AHOLD Hold Time	1.0 ns		95	
t <sub>50</sub>	AP Setup Time	5.0 ns		95	
t <sub>51</sub>	AP Hold Time	1.0 ns		95	
t <sub>52</sub>	BOFF# Setup Time	5.5 ns		95	
t <sub>53</sub>	BOFF# Hold Time	1.0 ns		95	
t <sub>54</sub>	BRDY# Setup Time	5.0 ns		95	
t <sub>55</sub>	BRDY# Hold Time	1.0 ns		95	
t <sub>56</sub>	BRDYC# Setup Time	5.0 ns		95	
t <sub>57</sub>	BRDYC# Hold Time	1.0 ns		95	
t <sub>58</sub>	D[63:0] Read Data Setup Time	2.8 ns		95	
t <sub>59</sub>	D[63:0] Read Data Hold Time	1.5 ns		95	
t <sub>60</sub>	DP[7:0] Read Data Setup Time	2.8 ns		95	
t <sub>61</sub>	DP[7:0] Read Data Hold Time	1.5 ns		95	
t <sub>62</sub>	EADS# Setup Time	5.0 ns		95	
t <sub>63</sub>	EADS# Hold Time	1.0 ns		95	
t <sub>64</sub>	EWBE# Setup Time	5.0 ns		95	
t <sub>65</sub>	EWBE# Hold Time	1.0 ns		95	
t <sub>66</sub>	FLUSH# Setup Time	5.0 ns		95	Note 2

**Notes:**

1. These level-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must be asserted for a minimum pulse width of two clocks.
2. These edge-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must have been negated at least two clocks prior to assertion and must remain asserted at least two clocks.

# Signal Switching Characteristics **16**

**Table 77. Input Setup and Hold Timings for 66-MHz Bus Operation (continued)**

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
$t_{67}$	FLUSH# Hold Time	1.0 ns		95	Note 2
$t_{68}$	HOLD Setup Time	5.0 ns		95	
$t_{69}$	HOLD Hold Time	1.5 ns		95	
$t_{70}$	IGNNE# Setup Time	5.0 ns		95	Note 1
$t_{71}$	IGNNE# Hold Time	1.0 ns		95	Note 1
$t_{72}$	INIT Setup Time	5.0 ns		95	Note 2
$t_{73}$	INIT Hold Time	1.0 ns		95	Note 2
$t_{74}$	INTR Setup Time	5.0 ns		95	Note 1
$t_{75}$	INTR Hold Time	1.0 ns		95	Note 1
$t_{76}$	INV Setup Time	5.0 ns		95	
$t_{77}$	INV Hold Time	1.0 ns		95	
$t_{78}$	KEN# Setup Time	5.0 ns		95	
$t_{79}$	KEN# Hold Time	1.0 ns		95	
$t_{80}$	NA# Setup Time	4.5 ns		95	
$t_{81}$	NA# Hold Time	1.0 ns		95	
$t_{82}$	NMI Setup Time	5.0 ns		95	Note 2
$t_{83}$	NMI Hold Time	1.0 ns		95	Note 2
$t_{84}$	SMI# Setup Time	5.0 ns		95	Note 2
$t_{85}$	SMI# Hold Time	1.0 ns		95	Note 2
$t_{86}$	STPCLK# Setup Time	5.0 ns		95	Note 1
$t_{87}$	STPCLK# Hold Time	1.0 ns		95	Note 1
$t_{88}$	WB/WT# Setup Time	4.5 ns		95	
$t_{89}$	WB/WT# Hold Time	1.0 ns		95	
<b>Notes:</b> <ol style="list-style-type: none"> <li>1. These level-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must be asserted for a minimum pulse width of two clocks.</li> <li>2. These edge-sensitive signals can be asserted synchronously or asynchronously. To be sampled on a specific clock edge, setup and hold times must be met. If asserted asynchronously, they must have been negated at least two clocks prior to assertion and must remain asserted at least two clocks.</li> </ol>					



# 16 Signal Switching Characteristics

## RESET and Test Signal Timing

Table 78. RESET and Configuration Signals for 100-MHz Bus Operation

Symbol	Parameter Description	Advance Data		Figure	Comments
		Min	Max		
t <sub>90</sub>	RESET Setup Time	1.7 ns		96	
t <sub>91</sub>	RESET Hold Time	1.0 ns		96	
t <sub>92</sub>	RESET Pulse Width, V <sub>CC</sub> and CLK Stable	15 clocks		96	
t <sub>93</sub>	RESET Active After V <sub>CC</sub> and CLK Stable	1.0 ms		96	
t <sub>94</sub>	BF[2:0] Setup Time	1.0 ms		96	Note 3
t <sub>95</sub>	BF[2:0] Hold Time	2 clocks		96	Note 3
t <sub>96</sub>	BRDYC# Hold Time	1.0 ns		96	Note 4
t <sub>97</sub>	BRDYC# Setup Time	2 clocks		96	Note 2
t <sub>98</sub>	BRDYC# Hold Time	2 clocks		96	Note 2
t <sub>99</sub>	FLUSH# Setup Time	1.7 ns		96	Note 1
t <sub>100</sub>	FLUSH# Hold Time	1.0 ns		96	Note 1
t <sub>101</sub>	FLUSH# Setup Time	2 clocks		96	Note 2
t <sub>102</sub>	FLUSH# Hold Time	2 clocks		96	Note 2

**Notes:**

1. To be sampled on a specific clock edge, setup and hold times must be met the clock edge before the clock edge on which RESET is sampled negated.
2. If asserted asynchronously, these signals must meet a minimum setup and hold time of two clocks relative to the negation of RESET.
3. BF[2:0] must meet a minimum setup time of 1.0 ms and a minimum hold time of two clocks relative to the negation of RESET.
4. If RESET is driven synchronously, BRDYC# must meet the specified hold time relative to the negation of RESET.

**Table 79. RESET and Configuration Signals for 66-MHz Bus Operation**

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
$t_{90}$	RESET Setup Time	5.0 ns		96	
$t_{91}$	RESET Hold Time	1.0 ns		96	
$t_{92}$	RESET Pulse Width, $V_{CC}$ and CLK Stable	15 clocks		96	
$t_{93}$	RESET Active After $V_{CC}$ and CLK Stable	1.0 ms		96	
$t_{94}$	BF[2:0] Setup Time	1.0 ms		96	Note 3
$t_{95}$	BF[2:0] Hold Time	2 clocks		96	Note 3
$t_{96}$	BRDYC# Hold Time	1.0 ns		96	Note 4
$t_{97}$	BRDYC# Setup Time	2 clocks		96	Note 2
$t_{98}$	BRDYC# Hold Time	2 clocks		96	Note 2
$t_{99}$	FLUSH# Setup Time	5.0 ns		96	Note 1
$t_{100}$	FLUSH# Hold Time	1.0 ns		96	Note 1
$t_{101}$	FLUSH# Setup Time	2 clocks		96	Note 2
$t_{102}$	FLUSH# Hold Time	2 clocks		96	Note 2

**Notes:**

1. To be sampled on a specific clock edge, setup and hold times must be met the clock edge before the clock edge on which RESET is sampled negated.
2. If asserted asynchronously, these signals must meet a minimum setup and hold time of two clocks relative to the negation of RESET.
3. BF[2:0] must meet a minimum setup time of 1.0 ms and a minimum hold time of two clocks relative to the negation of RESET.
4. If RESET is driven synchronously, BRDYC# must meet the specified hold time relative to the negation of RESET.

# 16 Signal Switching Characteristics

**Table 80. TCK Waveform and TRST# Timing at 25 MHz**

Symbol	Parameter Description	Preliminary Data		Figure	Comments
		Min	Max		
	TCK Frequency		25 MHz	97	
$t_{103}$	TCK Period	40.0 ns		97	
$t_{104}$	TCK High Time	14.0 ns		97	
$t_{105}$	TCK Low Time	14.0 ns		97	
$t_{106}$	TCK Fall Time		5.0 ns	97	Note 1, 2
$t_{107}$	TCK Rise Time		5.0 ns	97	Note 1, 2
$t_{108}$	TRST# Pulse Width	30.0 ns		98	Asynchronous

**Notes:**

1. Rise/Fall times can be increased by 1.0 ns for each 10 MHz that TCK is run below its maximum frequency of 25 MHz.
2. Rise/Fall times are measured between 0.8 V and 2.0 V.

**Table 81. Test Signal Timing at 25 MHz**

Symbol	Parameter Description	Preliminary Data		Figure	Notes
		Min	Max		
$t_{109}$	TDI Setup Time	5.0 ns		99	Note 2
$t_{110}$	TDI Hold Time	9.0 ns		99	Note 2
$t_{111}$	TMS Setup Time	5.0 ns		99	Note 2
$t_{112}$	TMS Hold Time	9.0 ns		99	Note 2
$t_{113}$	TDO Valid Delay	3.0 ns	13.0 ns	99	Note 1
$t_{114}$	TDO Float Delay		16.0 ns	99	Note 1
$t_{115}$	All Outputs (Non-Test) Valid Delay	3.0 ns	13.0 ns	99	Note 1
$t_{116}$	All Outputs (Non-Test) Float Delay		16.0 ns	99	Note 1
$t_{117}$	All Inputs (Non-Test) Setup Time	5.0 ns		99	Note 2
$t_{118}$	All Inputs (Non-Test) Hold Time	9.0 ns		99	Note 2

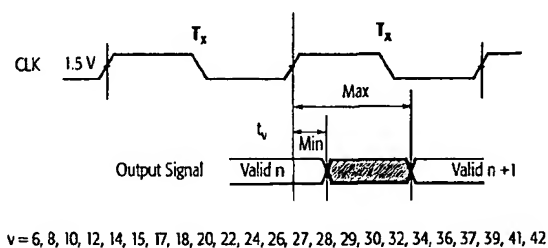
**Notes:**

1. Parameter is measured from the TCK falling edge.
2. Parameter is measured from the TCK rising edge.

# Signal Switching Characteristics **16**

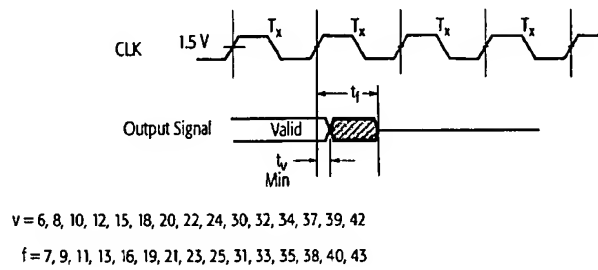
WAVEFORM	INPUTS	OUTPUTS
	Must be steady	Steady
	Can change from High to Low	Changing from High to Low
	Can change from Low to High	Changing from Low to High
	Don't care, any change permitted	Changing, State Unknown
	(Does not apply)	Center line is high impedance state

**Figure 92. Diagrams Key**

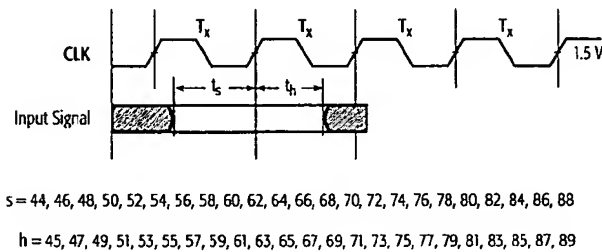


**Figure 93. Output Valid Delay Timing**

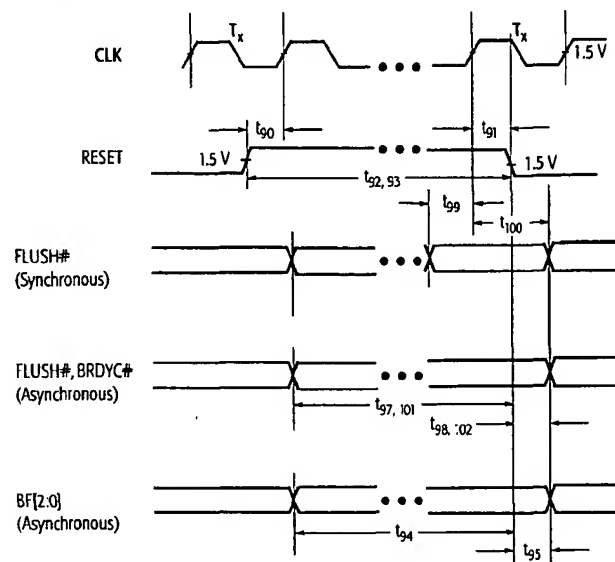
# 16 Signal Switching Characteristics



**Figure 94. Maximum Float Delay Timing**



**Figure 95. Input Setup and Hold Timing**



**Figure 96. Reset and Configuration Timing**

# 16 Signal Switching Characteristics

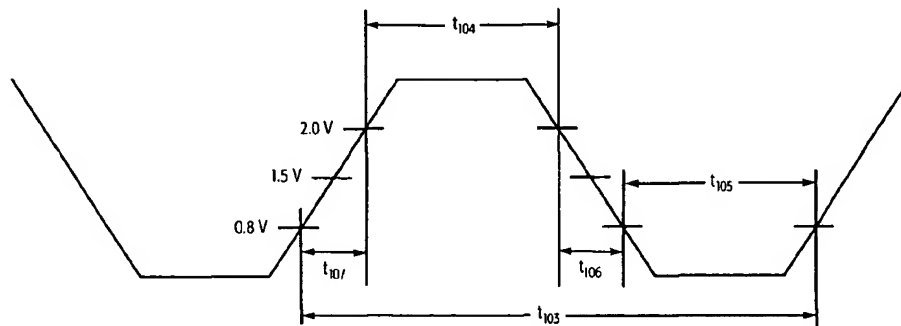


Figure 97. TCK Waveform

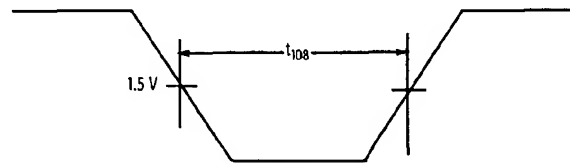


Figure 98. TRST# Timing

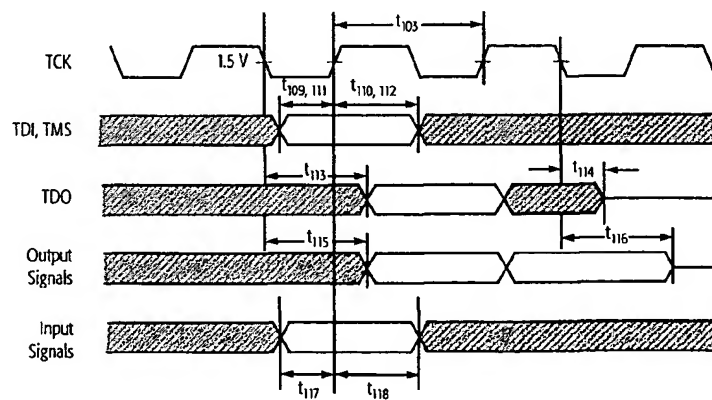


Figure 99. Test Signal Timing Diagram

## Thermal Design

This chapter contains thermal data that is subject to change. For the latest values, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).

### Package Thermal Specifications

The AMD-K6 3D processor operating specification calls for the case temperature ( $T_C$ ) to be in the range of 0°C to 70°C. The ambient temperature ( $T_A$ ) is not specified as long as the case temperature is not violated. The case temperature must be measured on the top center of the package. Table 82 shows the processor thermal specifications.

Table 82. Package Thermal Specification

$T_C$ Case Temperature	$\theta_{JC}$ Junction-Case	Maximum Thermal Power				
		233 MHz	250 MHz	266 MHz	300 MHz	333 MHz
0°C–70°C	1.7 °C/W	13.50 W	13.85 W	14.70 W	17.20 W	19.00 W
Stop Grant Mode		2.46 W	2.47 W	2.48 W	2.50 W	2.52 W
Stop Clock Mode		2.25 W	2.25 W	2.25 W	2.25 W	2.25 W



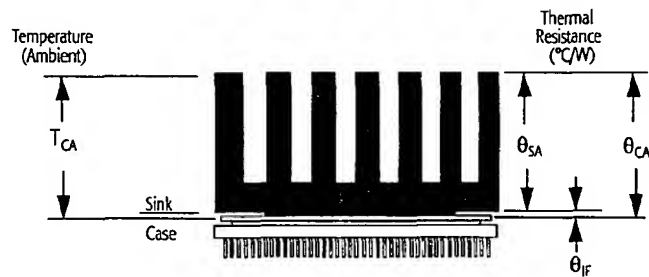
# 17 Thermal Design

Figure 100 shows the thermal model of a processor with a passive thermal solution. The case-to-ambient temperature ( $T_{CA}$ ) can be calculated from the following equation:

$$\begin{aligned} T_{CA} &= P_{MAX} \cdot \theta_{CA} \\ &= P_{MAX} \cdot (\theta_{IF} + \theta_{SA}) \end{aligned}$$

Where:

- $P_{MAX}$  = Maximum Power Consumption
- $\theta_{CA}$  = Case-to-Ambient Thermal Resistance
- $\theta_{IF}$  = Interface Material Thermal Resistance
- $\theta_{SA}$  = Sink-to-Ambient Thermal Resistance



**Figure 100. Thermal Model**

Figure 101 on page 333 illustrates the case-to-ambient temperature ( $T_{CA}$ ) in relation to the power consumption (X-axis) and the thermal resistance (Y-axis). If the power consumption and case temperature are known, the thermal resistance ( $\theta_{CA}$ ) requirement can be calculated for a given ambient temperature ( $T_A$ ) value.

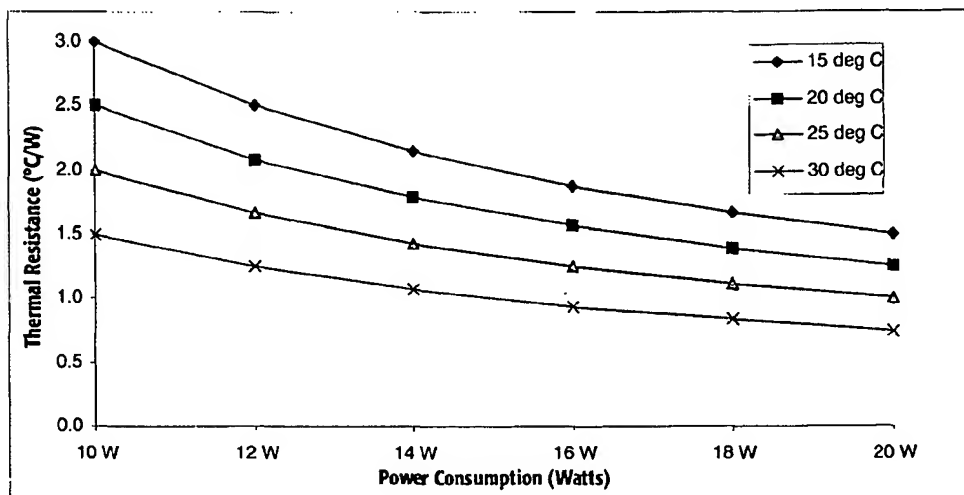


Figure 101. Power Consumption vs. Thermal Resistance

The following example calculates the required thermal resistance of a heatsink:

If:

$$T_C = 70^\circ\text{C}$$

$$T_A = 45^\circ\text{C}$$

$$P_{\text{MAX}} = 19.0\text{W at } 333\text{MHz}$$

Then:

$$\theta_{CA} \leq \left( \frac{T_C - T_A}{P_{\text{MAX}}} \right) = \frac{25^\circ\text{C}}{19.0\text{W}} = 1.32 (^\circ\text{C/W})$$

Thermal grease is recommended as interface material because it provides the lowest thermal resistance ( $\approx 0.20^\circ\text{C/W}$ ). The required thermal resistance ( $\theta_{SA}$ ) of the heatsink in this example is calculated as follows:

$$\theta_{SA} = \theta_{CA} - \theta_{IF} = 1.32 - 0.20 = 1.12 (^\circ\text{C/W})$$

# 17 Thermal Design

## Heat Dissipation Path

Figure 102 illustrates the processor's heat dissipation path. Most of the heat generated by the processor is dissipated from the top surface (ceramic and lid) of the package. The small amount of heat generated from the bottom side of the processor where the processor socket blocks the convection can be safely ignored.

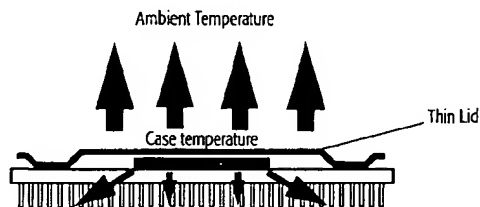


Figure 102. Processor Heat Dissipation Path

## Measuring Case Temperature

The case temperature must be measured on the top center of the package where most of the heat is dissipated. Figure 103 shows the correct location for measuring the case temperature. (If a heat exchange device is installed, the thermocouple must contact the processor top surface through a drilled hole.) The case temperature is measured to ensure that the thermal solution meets the operational specification.

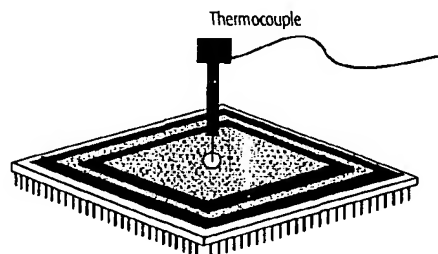


Figure 103. Measuring Case Temperature

## Layout and Airflow Considerations

### Voltage Regulator

A voltage regulator is required to support the lower voltage (3.3 V and lower) to the processor. In most applications, the voltage regulator is designed with power transistors. As a result, additional heatsinks are required to dissipate the heat from the power transistors. Figure 104 shows the voltage regulator placed parallel to the processor with the airflow aligned with the devices. With this alignment, the heat generated by the voltage regulator has minimal effect on the processor.

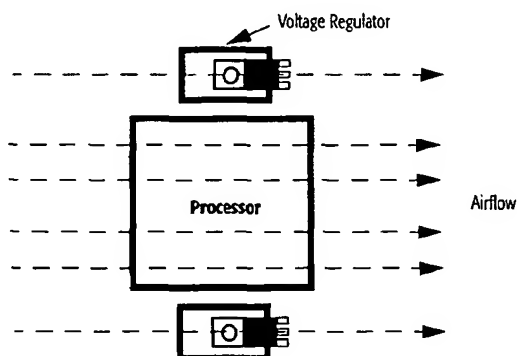


Figure 104. Voltage Regulator Placement

A heatsink and fan combination can deliver much better thermal performance than a heatsink alone. More importantly, with a fan/sink the airflow requirements in a system design are not as critical. A unidirectional heatsink with a fan moves air from the top of the heatsink to the side. In this case, the best location for the voltage regulator is on the side of the processor in the path of the airflow exiting the fan sink (see Figure 105 on page 336). This location guarantees that the heatsinks on both the processor and the regulator receive adequate air circulation.

## 17 Thermal Design

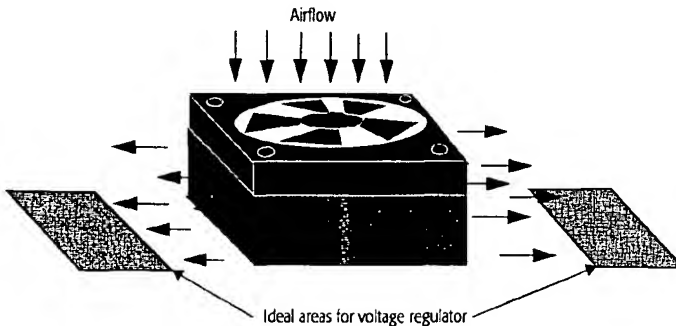


Figure 105. Airflow for a Heatsink with Fan

### Airflow Management in a System Design

Complete airflow management in a system is important. In addition to the volume of air, the path of the air is also important. Figure 106 shows the airflow in a dual-fan system. The fan in the front end pulls cool air into the system through intake slots in the chassis. The power supply fan forces the hot air out of the chassis. The thermal performance of the heatsink can be maximized if it is located in the shaded area, where it receives greatest benefit from this air exchange system.

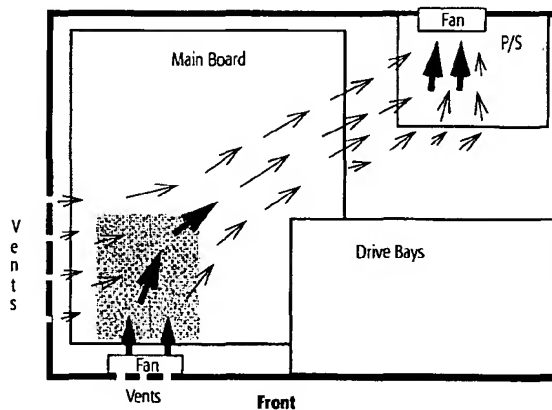
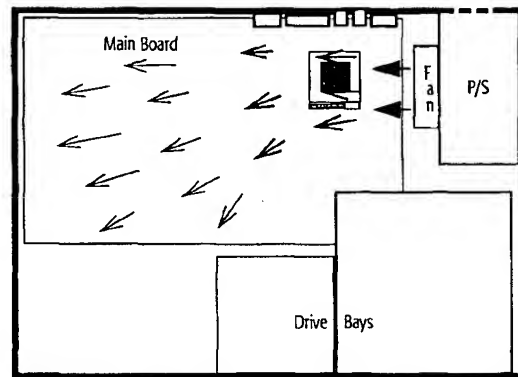


Figure 106. Airflow Path in a Dual-fan System

Figure 107 shows the airflow management in a system using the ATX form-factor. The orientation of the power supply fan and the motherboard are modified in the ATX platform design. The power supply fan pulls cool air through the chassis and across the processor. The processor is located near the power supply fan, where it can receive adequate airflow without an auxiliary fan. The arrangement significantly improves the airflow across the processor with minimum installation cost.



**Figure 107. Airflow Path in an ATX Form-Factor System**

---

## **17** *Thermal Design*

---

**338**

---

# 18

## Pins and Packaging

### Introduction

---

This chapter contains information about the AMD-K6 3D processor pin grid array, pin designations, and packaging. Pin placement is shown in a top-side view in Figure 108 on page 340 and in a pin-side view in Figure 109 on page 341. Table 83 on page 342 organizes the pins by functional grouping. Table 84 on page 343 gives the package specifications, which are illustrated in Figure 110 on page 344.

This chapter contains packaging information that is subject to change. For the latest information, see the AMD website at [www.amd.com/K6/k6docs/](http://www.amd.com/K6/k6docs/).



# 18 Pins and Packaging

## Pin Description Diagrams

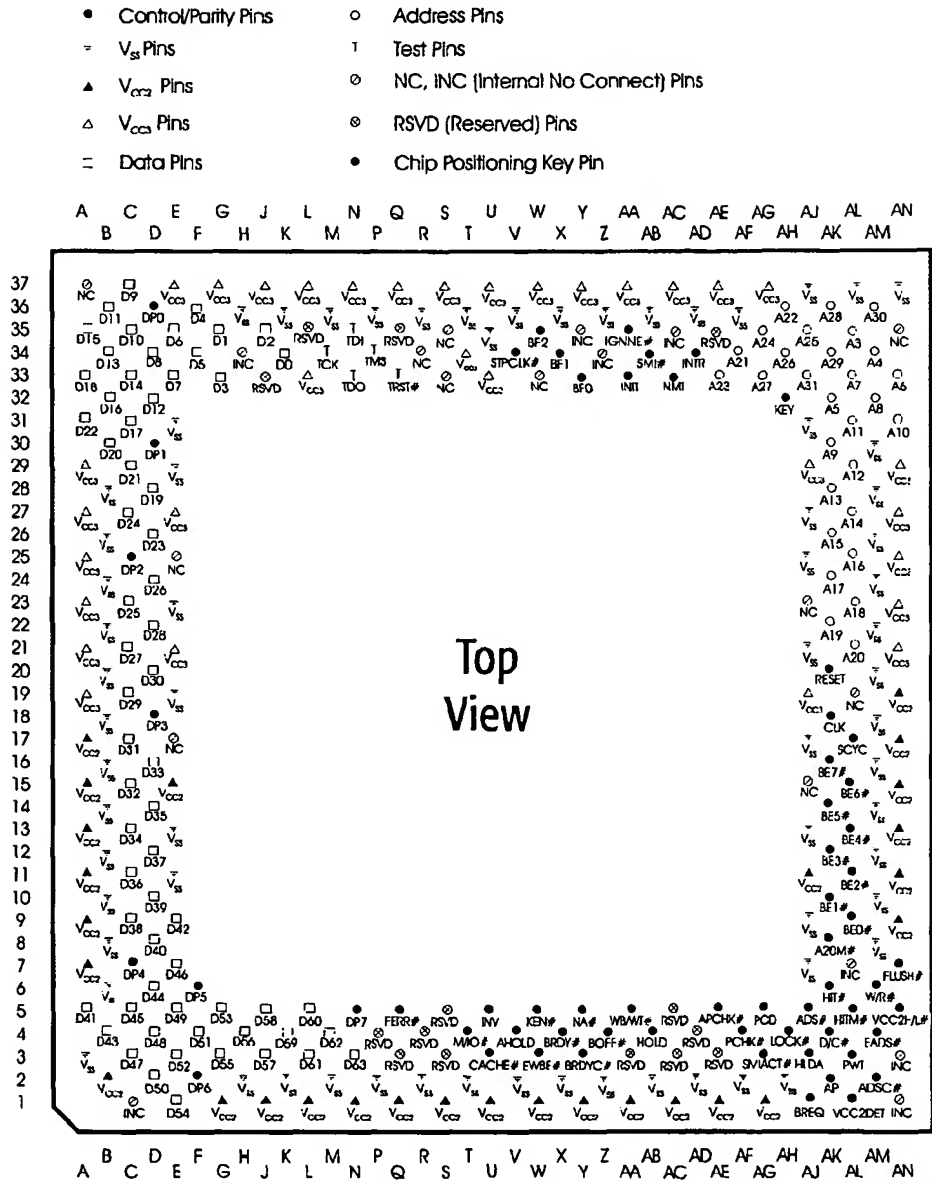


Figure 108. Processor Top-Side View

340

- 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37



341

# 18 Pins and Packaging

## Pin Designations

Table 83. Processor Functional Grouping

Address		Data		Control		Test		NC	V <sub>cc2</sub>	V <sub>cc3</sub>	V <sub>ss</sub>
Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin No.	Pin No.	Pin No.	Pin No.
A3	AL-35	D0	K-34	A20M#	AK-08	TCK	M-34	A-37	A-07	A-19	A-03
A4	AM-34	D1	G-35	ADS#	AJ-05	TDI	N-35	E-17	A-09	A-21	B-06
A5	AK-32	D2	J-35	ADSC#	AM-02	TDO	N-33	E-25	A-11	A-23	B-08
A6	AN-33	D3	G-33	AHOLD	V-04	TMS	P-34	R-34	A-13	A-25	B-10
A7	AL-33	D4	F-36	APCHK#	AE-05	TRST#	Q-33	S-33	A-15	A-27	B-12
A8	AM-32	D5	F-34	BE0#	AL-09			S-35	A-17	A-29	B-14
A9	AK-30	D6	E-35	BE1#	AK-10			W-33	B-02	E-21	B-16
A10	AN-31	D7	E-33	BE2#	AL-11			AI-15	E-15	E-27	B-18
A11	AL-31	D8	D-34	BE3#	AK-12			AI-23	G-01	E-37	B-20
A12	AL-29	D9	C-37	BE4#	AL-13			AI-19	J-01	G-37	B-22
A13	AK-28	D10	C-35	BE5#	AK-14	AP	AK-02	AN-35	L-01	J-37	B-24
A14	AL-27	D11	B-36	BE6#	AL-15	DP0	D-36		N-01	L-33	B-26
A15	AK-26	D12	D-32	BE7#	AK-16	DP1	D-30		Q-01	L-37	B-28
A16	AL-25	D13	B-34	BF0	Y-33	DP2	C-25	INC	S-01	N-37	E-11
A17	AK-24	D14	C-33	BF1	X-34	DP3	D-18		U-01	Q-37	E-13
A18	AL-23	D15	A-35	BF2	W-35	DP4	C-07		W-01	S-37	E-19
A19	AK-22	D16	B-32	BOFF#	Z-04	DP5	F-06	C-01	Y-01	T-34	E-23
A20	AL-21	D17	C-31	BRDY#	X-04	DP6	F-02	H-34	AA-01	U-33	E-29
A21	AF-34	D18	A-33	BRDYC#	Y-05	DP7	N-05	Y-35	AC-01	U-37	E-31
A22	AH-36	D19	D-28	BREQ	AJ-01			Z-34	AE-01	W-37	H-02
A23	AE-33	D20	B-30	CACHE#	U-03			AC-35	AG-01	Y-37	H-36
A24	AG-35	D21	C-29	CLK	AK-18			AL-07	AJ-11	AA-37	K-02
A25	AJ-35	D22	A-31	D/C#	AK-04			AN-01	AN-09	AC-37	K-36
A26	AH-34	D23	D-26	EADS#	AM-04			AN-03	AN-11	AE-37	M-02
A27	AG-33	D24	C-27	EWBE#	W-03				AN-13	AG-37	M-36
A28	AK-36	D25	C-23	FERR#	Q-05				AN-15	AJ-19	P-02
A29	AK-34	D26	D-24	FLUSH#	AN-07			RSVD	AJ-19	AJ-29	P-36
A30	AM-36	D27	C-21	HIT#	AK-06				AN-17	AN-21	R-02
A31	AJ-33	D28	D-22	HITM#	AL-05					AN-23	R-36
		D29	C-19	HLDA	AJ-03			J-33		AN-25	T-02
		D30	D-20	HOLD	AB-04			L-35		AN-27	T-36
		D31	C-17	IGNNE#	AA-35			P-04		AN-29	U-35
		D32	C-15	INIT	AA-33			Q-03			V-02
		D33	D-16	INTR	AD-34			Q-35			V-36
		D34	C-13	INV	U-05			R-04			X-02
		D35	D-14	KEN#	W-05			S-03			X-36
		D36	C-11	LOCK#	AH-04			S-05			Z-02
		D37	D-12	M/O#	T-04			AA-03			Z-36
		D38	C-09	NA#	Y-05			AC-03			AB-02
		D39	D-10	NMI	AC-33			AC-05			AB-36
		D40	D-08	PCD	AG-05			AD-04			AD-02
		D41	A-05	PCHK#	AF-04			AE-03			AD-36
		D42	E-09	PWT	AL-03			AE-35			AF-02
		D43	B-04	RESET	AK-20						AF-36
		D44	D-06	SCYC	AL-17						AH-02
		D45	C-05	SMT#	AB-34			KEY			AJ-07
		D46	E-07	SMACT#	AG-03						AJ-09
		D47	C-03	STPCLK#	V-34			AH-32			AJ-13
		D48	D-04	VCCDET	AL-01						AJ-17
		D49	E-05	VCC2/VL#	AN-05						AJ-21
		D50	D-02	VW/R#	AM-06						AJ-25
		D51	F-04	VW/WT#	AA-05						AJ-27
		D52	E-03								AJ-31
		D53	G-05								AJ-37
		D54	E-01								AL-37
		D55	G-03								AM-08
		D56	H-04								AM-10
		D57	J-03								AM-12
		D58	J-05								AM-14
		D59	K-04								AM-16
		D60	L-05								AM-18
		D61	L-03								
		D62	M-04								
		D63	N-03								

## Package Specifications

### 321-Pin Staggered CPGA Package Specification

Table 84. 321-Pin Staggered CPGA Package Specification

Symbol	Millimeters			Inches		
	Min	Max	Notes	Min	Max	Notes
A	49.28	49.78		1.940	1.960	
B	45.59	45.85		1.795	1.805	
C	31.32	32.59		1.233	1.283	
D	44.90	45.10		1.768	1.776	
E	2.91	3.63		0.115	0.143	
F	1.30	1.52		0.051	0.060	
G	3.05	3.30		0.120	0.130	
H	0.43	0.51		0.017	0.020	
M	2.29	2.79		0.090	0.110	
N	1.14	1.40		0.045	0.055	
d	1.52	2.29		0.060	0.090	
e	1.52	2.54		0.060	0.100	
f	—	0.13	Flatness	—	0.005	Flatness

# 18 Pins and Packaging

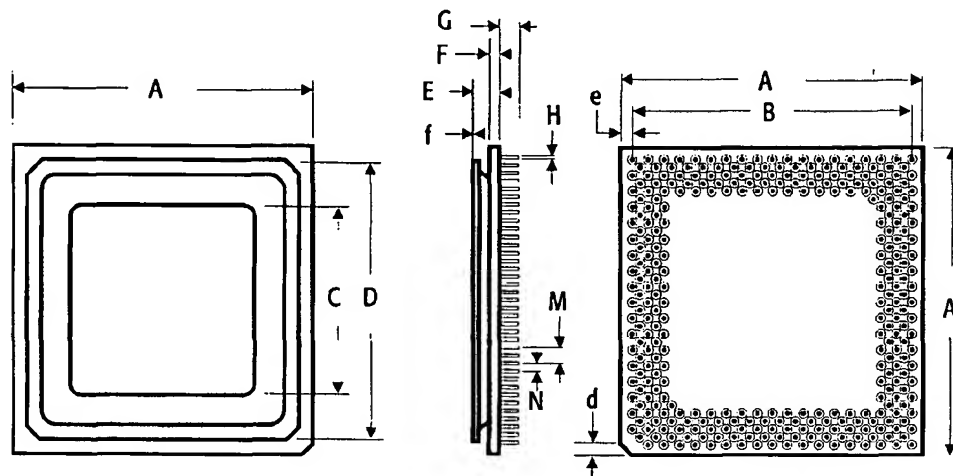


Figure 110. 321-Pin Staggered CPGA Package Specification

# ***19***

## **Ordering Information**

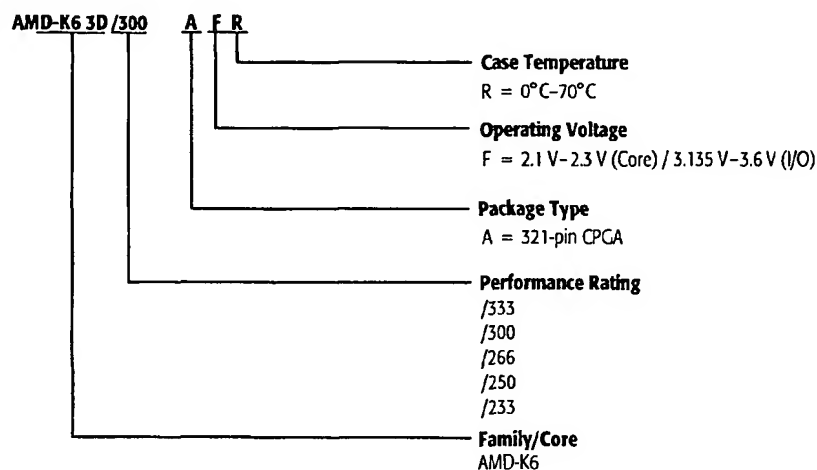
The ordering information contained in this chapter is subject to change. For the latest information, see the AMD website at [www.amd.com/k6/k6docs/](http://www.amd.com/k6/k6docs/).

***345***

# 19 Ordering Information

## Standard AMD-K6 3D Processor Model 8 Products

AMD standard products are available in several operating ranges. The ordering part number (OPN) is formed by a combination of the elements below.



**Table 85. Valid Ordering Part Number Combinations**

OPN	Package Type	Operating Voltage	Case Temperature
AMD-K6 3D/333AFR	321-pin CPGA	2.1 V–2.3 V (Core) 3.135 V–3.6 V (I/O)	0°C–70°C
AMD-K6 3D/300AFR	321-pin CPGA	2.1 V–2.3 V (Core) 3.135 V–3.6 V (I/O)	0°C–70°C
AMD-K6 3D/266AFR	321-pin CPGA	2.1 V–2.3 V (Core) 3.135 V–3.6 V (I/O)	0°C–70°C
AMD-K6 3D/250AFR	321-pin CPGA	2.1 V–2.3 V (Core) 3.135 V–3.6 V (I/O)	0°C–70°C
AMD-K6 3D/233AFR	321-pin CPGA	2.1 V–2.3 V (Core) 3.135 V–3.6 V (I/O)	0°C–70°C
<b>Note:</b> This table lists configurations planned to be supported in volume for this device. Consult the local AMD sales office to confirm availability of specific valid combinations and to check on newly-released combinations.			

# ***Appendix A***

## **MMX Multimedia Technology**

### **Introduction**

---

MMX multimedia technology was originally introduced in the AMD-K6 processor, before the development of 3D technology. The AMD-K6 3D processor executes both MMX and 3D instruction sets. References to multimedia technology in this appendix pertain to the MMX multimedia technology. For information on 3D technology, see Chapter 4, “3D Technology” on page 81. References to the AMD-K6 3D processor in this appendix also apply to the AMD-K6 processor.

PC performance requirements are being driven by emerging multimedia and communications software. 3D graphics, video, audio, and telephony capabilities are evolving across education, entertainment, and internet applications. As multimedia applications continue to proliferate in the marketplace, PC systems suppliers are being challenged to deliver multimedia-enabled PC solutions at reasonable prices.

The AMD-K6 3D processor incorporates a robust multimedia technology that is fully software compatible with the MMX technology as defined by Intel. This multimedia technology enables scaleable multimedia capabilities across a broad range of PC systems.



---

# **A** *MMX Multimedia Technology*

---

The processor features a decode-decoupled superscalar microarchitecture and state-of-the-art design techniques to deliver true sixth-generation performance while maintaining full x86 binary software compatibility. An x86 binary-compatible processor implements the industry-standard x86 instruction set by decoding and executing the x86 instruction set as its native mode of operation. Only this native mode enables delivery of maximum performance when running PC software.

The processor delivers leading-edge performance to mainstream PC systems running industry-standard x86 software. The processor implements advanced design techniques like instruction pre-decoding, dual x86 opcode decoding, single-cycle internal RISC operations, parallel execution units, out-of-order execution, data forwarding, register renaming, and dynamic branch prediction. In other words, the AMD-K6 3D processor is capable of issuing, executing, and retiring multiple x86 instructions per cycle, resulting in superior scaleable performance.

This appendix describes the multimedia technology of the processor, including data types, instructions, and programming considerations.

## **MMX Multimedia Technology Architecture**

---

The multimedia technology in the processor is designed to accelerate media and communication applications. Specialized applications that use music synthesis, speech synthesis, speech recognition, audio and video compression and decompression, full motion video, 2D and 3D graphics, and video conferencing, can take advantage of the AMD-K6 3D processor multimedia technology. The multimedia technology implements new instructions, new data types, and powerful parallel processing (Single Instruction Multiple Data—SIMD) techniques that can significantly increase the performance of these applications.

### **Key Functionality**

At the lowest levels, multimedia applications (audio, video, 3D graphics, and telephony, etc.) contain many similar functions. When these functions are performed on a processor that does not have MMX capability, the processor is heavily burdened by

the computational requirements of this information. Processors executing the MMX instructions increase the performance of multimedia applications. This performance increase is a direct result of the increased multimedia bandwidth of the processor.

Multimedia applications must process large amounts of data. Parallel data computing is exemplified by applications that manipulate screen pixel information. Instead of acting on one pixel at a time, multimedia technology enables the system to act on multiple pixels simultaneously. This SIMD model is a key feature of MMX technology.

The AMD-K6 3D processor multimedia technology architecture includes four new MMX data types, 57 new MMX instructions, eight new 64-bit MMX registers, and an SIMD processing pipeline. The multimedia technology is compatible with existing x86 applications.

The 57 new MMX instructions include arithmetic functions, packing and unpacking functions, logical operations, and moves. These are the basic functions that are most commonly used in repetitive computational multimedia programs.

Multimedia applications often use smaller operands—8-bit data is commonly used for pixel information and 16-bit data is used for audio samples. The new MMX registers allow data to be packed into 64-bit operands. For example, 8-bit data (1 byte) can be packed in sets of eight in a single 64-bit register, and all eight bytes can be operated on simultaneously by a single MMX instruction.

For 256-color video modes, this translates to computing eight pixels per instruction. When an entire screen is being re-drawn, these pixel manipulation routines often use highly repetitive loops. Parallel processing of eight pieces of data can reduce the processing time of a code loop by up to a factor of eight.

Multimedia applications frequently multiply and accumulate data. The multimedia technology provides instructions that add, multiply, and even combine these operations. For example, the PMADDWD instruction can multiply and then add words of data in a single instruction that uses far less processor cycles than the equivalent x86 operations.

# **A** *MMX Multimedia Technology*

## **Executing MMX Instructions**

A programmer must approach the use of MMX instructions differently, based on whether the code being developed is at the system level or at the application level. The details of these differences are discussed in “MMX Programming Considerations” on page 355.

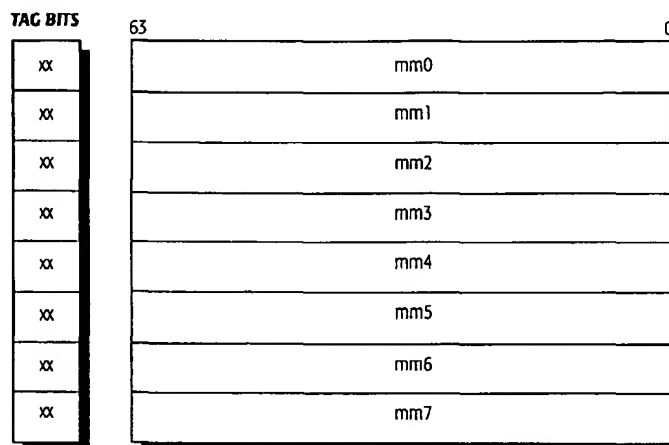
Before using the MMX instructions, the programmer must use the CUID instruction to determine if the processor supports multimedia technology. For more information, see Appendix C, “AMD Processor Recognition” on page 505.

Function 1 (EAX=1) of the processor CUID instruction returns the processor feature bits in the EDX register. Software can then test bit 23 of the feature bits to determine if the processor supports the multimedia technology. If bit 23 is set to 1, MMX instructions are supported. All AMD-K6 3D processors have bit 23 set. Once it is determined that multimedia technology is supported, subsequent code can use the MMX instructions. Alternatively, the AMD 8000\_0001h extended CUID function can be used to test whether the processor supports multimedia technology.

After a module of MMX code has executed, the programmer must empty the MMX state by executing the EMMS command. Because the MMX registers share the floating-point registers, an instruction is needed to prevent MMX code from interfering with floating-point. The EMMS command clears the multimedia state and resets all the floating-point tag bits. Emptying the MMX state sets the floating-point tag bits to empty (all 1s), which marks the MMX/FP registers as invalid and available.

## **MMX Register Set**

The AMD-K6 3D processor implements eight new 64-bit MMX registers. These registers are mapped on the floating-point registers. As shown in Figure 111 on page 351, the new MMX instructions refer to these registers as mm0 to mm7. Mapping the new MMX registers on the floating-point stack enables backwards compatibility for the register saving that must occur as a result of task switching.



**Figure 111. MMX Registers**

Aliasing the MMX registers onto the floating-point stack registers provides a safe way to introduce this new technology. Instead of needing to modify operating systems, new MMX applications can be supported through device drivers, MMX libraries, or DLL files. See “MMX Programming Considerations” on page 355 for more information.

Current operating systems have support for floating-point operations. Using the floating-point registers for MMX code is an ingenious way of implementing automatic support for MMX instructions. Every time the processor executes an MMX instruction, all the floating-point register tag bits are set to zero (00b=valid). Setting the tag bits after every MMX instruction prevents the processor from having to perform extra tasks. These extra tasks are normally executed on floating-point registers when the Tag field is something other than 00b.

If a task switch occurs during an MMX or floating-point instruction, the Control Register (CR0) Task Switch (TS) bit is set to 1. The processor then generates an interrupt 7 (int 7 Device Not Available) when it encounters the next floating-point or MMX instruction, allowing the operating system to save the state of the MMX/FP registers.

# **A** *MMX Multimedia Technology*

If there is a task switch when MMX applications are running with older applications that do not include MMX instructions, the MMX/FP register state is still saved automatically through the int 7 handler.

## **MMX Data Type Details**

The processor multimedia technology uses a packed data format. The data is packed in a single, 64-bit MMX register or memory operand as eight bytes, four words, or two double words. Each byte, word, doubleword, or quadword is an integer data type.

The form of an instruction determines the data type. For example, the MOV instruction comes in two different forms—MOVD moves 32 bits of data and MOVQ moves 64 bits of data.

The four new data types are defined as follows:

<u>Packed byte</u>	Eight 8-bit bytes packed into 64 bits Signed integer range( $-2^7$ to $2^7-1$ ) Unsigned integer range(0 to $2^8-1$ )
<u>Packed word</u>	Four 16-bit words packed into 64-bits Signed integer range( $-2^{15}$ to $2^{15}-1$ ) Unsigned integer range(0 to $2^{16}-1$ )
<u>Packed doubleword</u>	Two 32-bit doublewords packed into 64 bits Signed integer range( $-2^{31}$ to $2^{31}-1$ ) Unsigned integer range(0 to $2^{32}-1$ )
<u>Quadword</u>	One 64-bit quadword Signed integer range( $-2^{63}$ to $2^{63}-1$ ) Unsigned integer range(0 to $2^{64}-1$ )

Figure 112 on page 353 shows the four new data types.

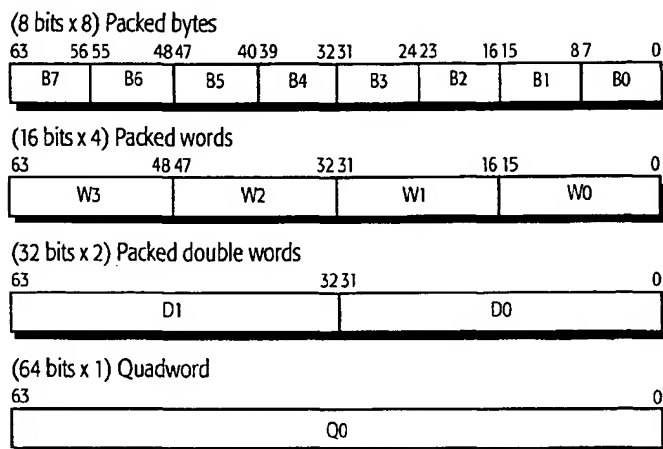


Figure 112. MMX Data Types

## MMX Instructions

The processor multimedia technology includes 57 new MMX instructions. These instructions are organized into the following groups:

- Arithmetic
- Empty MMX registers
- Compare
- Convert (pack/unpack)
- Logical
- Move
- Shift

The following mnemonics are used in the instructions:

- P—Packed data
- B—Byte
- W—Word
- D—Doubleword
- Q—Quadword
- S—Signed

# A MMX Multimedia Technology

- U—Unsigned
- SS—Signed Saturation
- US—Unsigned Saturation

For example, the mnemonic for the PACK instruction that packs four words into eight unsigned bytes is PACKUSWB. In this mnemonic, the US designates an unsigned result with saturation, and the WB means that the source is packed words and the result is packed bytes.

The term *saturation* is commonly used in multimedia applications. Saturation allows mathematical limits to be placed on the data elements. If a result exceeds the boundary of that data type, the result is set to the defined limit for that instruction. A common use of saturation is to prevent color wraparound.

## MMX Instruction Formats

All MMX instructions, except the EMMS instruction that uses no operands, are formatted as follows:

```
INSTRUCTION mmreg1, mmreg2/mem64
```

The source operand (mmreg2/mem64) can be either an MMX register or a memory location. The destination operand (mmreg1) can only be an MMX register.

The MOVD and MOVQ instructions also have the following acceptable formats:

```
MOVD      mreg32/mem32, mmreg1
MOVD      mmreg1, mreg32/mem32
MOVQ      mem64, mmreg1
```

In the first example, the source operand (mreg32/mem32) can be either an integer register or a 32-bit memory address. The destination operand (mmreg1) can only be an MMX register. The second example has the source operand as an MMX register. The destination operand (mreg32/mem32) can be either an integer register or a 32-bit memory address. The third example has the source operand as an MMX register and the destination operand as a 64-bit memory location.

The SHIFT instructions can also utilize an immediate source operand. It is designated as *imm8*.

```
PSRLW      mmreg1, imm8
```

## MMX Programming Considerations

---

This section describes considerations for programmers writing operating systems, compilers, and applications that utilize MMX instructions as implemented in the AMD-K6 3D processor.

### MMX Feature Detection

To use the AMD-K6 3D processor MMX multimedia technology, the programmer must determine if the processor supports it. The CUID instruction gives programmers the ability to determine the presence of multimedia technology on the processor. Software must first test to see if the CUID instruction is supported. For a detailed description of the CUID instruction, see Appendix C, “AMD Processor Recognition” on page 505.

The presence of the CUID instruction is indicated by the ID bit (21) in the EFLAGS register. See “Testing for the CUID Instruction” on page 506 for more information.

If the processor supports the CUID instruction, the programmer must execute the standard function, EAX=0. The CUID function returns a 12-character string that identifies the processor’s vendor. For AMD processors, standard function 0 returns a vendor string of “Authentic AMD”. This string requires the software to follow the AMD definitions for subsequent CUID functions and the values returned for those functions.

The next step is for the programmer to determine if MMX instructions are supported. Function 1 of the CUID instruction provides this information. Function 1 (EAX=1) of the AMD CUID instruction returns the feature bits in the EDX register. If bit 23 in the EDX register is set to 1, MMX instructions are supported. The following code sample shows how to test for MMX instruction support.

```
mov    eax,1           ; setup function 1
CUID   ; call the function
test   edx, 800000      ; test 23rd bit
jnz    YES_MM           ; multimedia technology supported
```



# A MMX Multimedia Technology

Alternatively, the extended function 1 (EAX=8000\_0001h) can be used to determine if MMX instructions are supported.

```
mov    eax,8000_0001h    ; setup extended function 1
CUID   ; call the function
test   edx, 800000        ; test 23rd bit
jnz    YES_MM             ; multimedia technology supported
```

## Task Switching

A task switch is an event that occurs within operating systems that allows multiple programs to be executed in parallel. Most modern operating systems utilizing task switching are called multitasking operating systems.

There are two types of multitasking operating systems—cooperative and preemptive.

### Cooperative Multitasking

In cooperative multitasking operating systems, applications do not care about other tasks that may be running. Each task assumes that it owns the machine state (processor, registers, I/O, memory, etc.). In addition, these tasks must take care of saving their own information (i.e., registers, stacks, states) in their own memory areas. The cooperative multitasking operating system does not save operating state information for the applications.

There are different types of cooperative multitasking operating systems. Some of these operating systems perform some level of state saves, but this state saving is not always reliable. All software engineers programming for a cooperative multitasking environment must save the MMX or floating-point states before relinquishing control to another task or to the operating system. The FSAVE and FRSTOR commands are used to perform this task. Figure 113 on page 357 illustrates this task switching process.

*Note: Some cooperative operating systems may have API calls to perform these tasks for the application.*

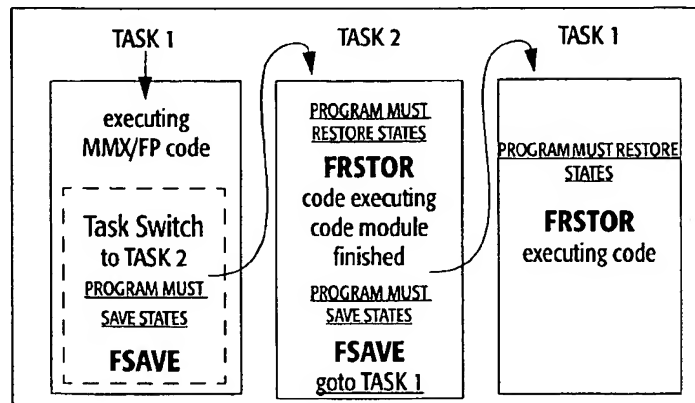


Figure 113. Cooperative Task Switching

#### Preemptive Multitasking

In preemptive multitasking operating systems like OS/2, Windows NT, and UNIX, the operating system handles all state and register saves. The application programmer does not need to save states when programming within a preemptive multitasking environment. The preemptive multitasking operating system sets aside a save area for each task.

In a preemptive multitasking operating system, if a task switch occurs, the operating system sets the Control Register 0 (CR0) Task Switch (TS) bit to 1. If the new task encounters a floating-point or MMX instruction, an interrupt 7 (int 7, Device Not Available) is generated. The int7 handler saves the state of the first task and restores the state of the second task. The int7 handler sets the CR0.TS to 0 and returns to the original floating-point or MMX instruction in the second task. Figure 114 illustrates this task switching process.

# A MMX Multimedia Technology

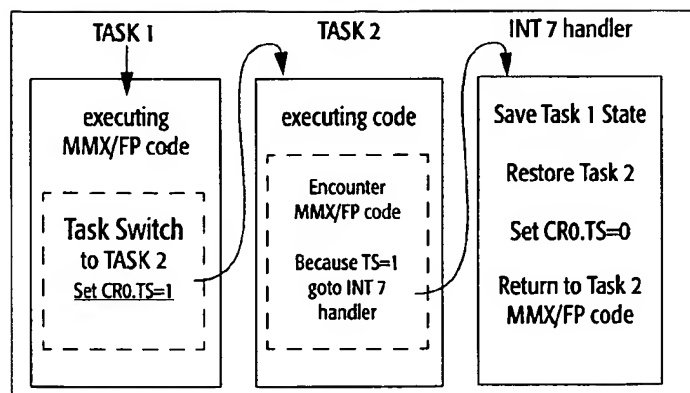


Figure 114. Preemptive Task Switching

## MMX Exceptions

There are no new exceptions defined for supporting the MMX and 3D instructions. All exceptions that occur while decoding or executing an MMX or 3D instruction are handled in existing exception handlers without modification. See “3D Exceptions” on page 93 for more information.

## Mixing MMX and Floating-Point Instructions

The programmer must take care when writing code that contains both MMX and floating-point instructions. The MMX code modules should be separated from the floating-point code modules. All code of one type (MMX or floating-point code) should be grouped together as often as possible. To obtain the highest performance, routines should not contain any conditional branches at the end of loops that jump to code of a different type than the code that is currently being executed.

In certain multimedia environments, floating-point and MMX instructions may be mixed. For example, if a programmer wants to change the viewing perspective of a three-dimensional scene, the perspective can be changed through transformation matrices using floating-point registers. The picture/pixel information is integer-based and requires MMX instructions to

manipulate this information. Both MMX and floating-point instructions are required to perform this task.

The software must clean up after itself at the end of an MMX code module. The EMMS instruction must be used at the end of an MMX code module to mark all floating-point registers as empty (11=empty/invalid). In cooperative multitasking operating systems, the EMMS instruction must be used when switching between tasks.

*Note: In some situations, experienced programmers can utilize the MMX registers to pass information between tasks. In these situations, the EMMS instruction is not required.*

The tag bits are affected by every MMX and floating-point instruction. After every MMX instruction except EMMS, all the tag bits in the floating-point tag word are set to 0. When the EMMS instruction is executed, all the tag bits in the tag word are set to 1. For more information, see "Floating-Point and MMX/3D Instruction Compatibility" on page 256.

## Prefixes

All instructions in the x86 architecture translate to a binary value or opcode. This 1 or 2 byte opcode value is different for each instruction. If an instruction is two bytes long, the second byte is called the Mod R/M byte. The Mod R/M byte is used to further describe the type of instruction that is used.

The x86 opcode and the Mod R/M byte can also be followed by an SIB byte. This byte is used to describe the Scale, Index and Base forms of 32-bit addressing.

The format of the x86 instruction allows for certain prefixes to be placed before each instruction. These prefixes indicate different types of command overrides.

The MMX instructions follow these rules just like all the current existing instructions. This allows for an easy implementation into the x86 architecture. All of the rules that apply to the x86 architecture apply to MMX instructions, including accessing registers, memory, and I/O.

---

# **A** *MMX Multimedia Technology*

---

Most opcode prefixes can be utilized while using MMX instructions. The following prefixes can be used with MMX instructions:

- The Segment Override prefixes (2Eh/CS, 36h/SS, 3Eh/DS, 26h/ES, 64h/FS, and 65h/GS) affect MMX instructions that contain a memory operand.
- The LOCK prefix (F0h) triggers an invalid opcode exception (interrupt 6).
- The Address Size Override prefix (67h) affects MMX instructions that contain a memory operand.

## **MMX Instruction Set**

---

The following MMX instruction definitions are in alphabetical order according to the instruction mnemonics.

## EMMS

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
EMMS	0F 77h	Clear the MMX state

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.

The EMMS instruction is used to clear the MMX state following the execution of a block of code using MMX instructions. Because the MMX registers and tag words are shared with the floating-point unit, it is necessary to clear the state before executing code that includes floating-point instructions.

# A MMX Multimedia Technology

## MOVD

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
MOVD mmreg1, reg32/mem32	0F 6Eh	Copy a 32-bit value from the general purpose register or memory location into the MMX register
MOVD reg32/mem32, mmreg1	0F 7Eh	Copy a 32-bit value from the MMX register into the general purpose register or memory location

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The MOVD instruction moves a 32-bit data value from an MMX register to a general purpose register or memory, or it moves the 32-bit data from a general purpose register or memory into an MMX register. If the 32-bit data to be moved is provided by an MMX register, the instruction moves bits 31–0 of the MMX register into the specified register or memory location. If the 32-bit data is being moved into an MMX register, the instruction moves the 32-bits of data into bits 31–0 of the MMX register and fills bits 63–32 with zeros.

**Related Instructions** See the MOVQ instruction.

## MOVQ

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
MOVQ mmreg1, mmreg2/mem64	0F 6Fh	Copy a 64-bit value from an MMX register or memory location into an MMX register
MOVQ mmreg2/mem64, mmreg1	0F 7Fh	Copy a 64-bit value from an MMX register into an MMX register or memory location

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The MOVQ instruction moves a 64-bit data value from one MMX register to another MMX register or memory, or it moves the 64-bit data from one MMX register or memory to another MMX register. Copying data from one memory location to another memory location cannot be accomplished with the MOVQ instruction.

**Related Instructions** See the MOVD instruction.



# A MMX Multimedia Technology

## PACKSSDW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PACKSSDW mmreg1, mmreg2/mem64	0F 6Bh	Pack with saturation signed 32-bit operands into signed 16-bit results

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

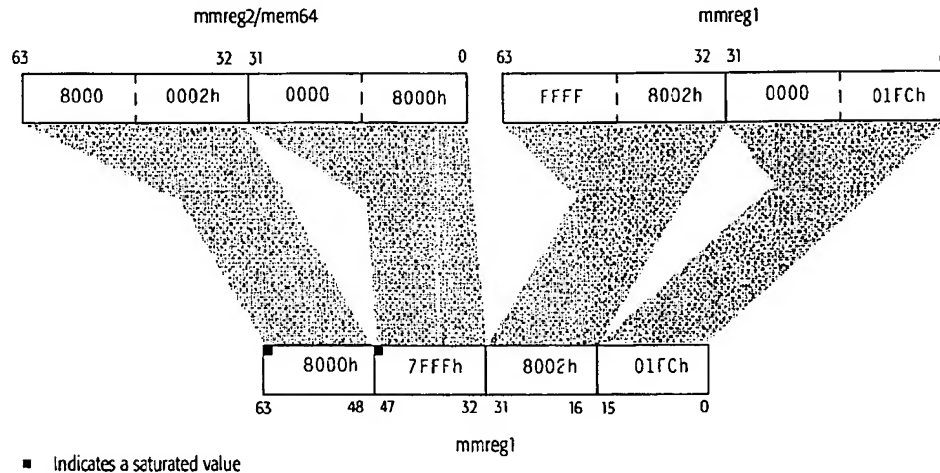
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PACKSSDW instruction performs a pack and saturate operation on two signed 32-bit values in the first operand and two signed 32-bit values in the second operand. The four signed 16-bit results are placed in the specified MMX register.

The pack operation is a data conversion. The PACKSSDW instruction converts or packs the four signed 32-bit values into four signed 16-bit values, applying saturating arithmetic. If the signed 32-bit value is less than -32768 (8000h), it saturates to -32768 (8000h). If the signed 32-bit value is greater than 32767 (7FFFh), it saturates to 32767 (7FFFh). All values between -32768 and 32767 are represented with their signed 16-bit value.

The first operand must be an MMX register. In addition to providing the first operand, this MMX register is the location where the result of the pack and saturate operation is stored. The second operand can be an MMX register or a 64-bit memory location.

## Functional Illustration of the PACKSSDW Instruction



The following list explains the functional illustration of the PACKSSDW instruction:

- Bits 63–32 of the source operand (**mmreg2/mem64**) are packed into bits 63–48 of the destination operand (**mmreg1**). The result is saturated to the largest possible 16-bit negative number because the 32-bit negative source operand (8000\_0002h) exceeds the capacity of the signed 16-bit destination operand.
- Bits 31–0 of the source operand are packed into bits 47–32 of the destination operand. The result is saturated to the largest possible 16-bit positive number because the 32-bit positive source operand (0000\_8000h) exceeds the capacity of the 16-bit destination operand.
- Bits 63–32 of the destination operand are packed into bits 31–16 of the destination operand. The results are not saturated because the 32-bit negative source operand (FFFF\_8002h) does not exceed the capacity of the 16-bit destination operand.
- Bits 31–0 of the destination operand are packed into bits 15–0 of the destination operand. The results are not saturated because the 32-bit positive source operand (0000\_01FCh) does not exceed the capacity of the 16-bit destination operand.

### Related Instructions

See the PACKSSWB instruction.  
 See the PACKUSWB instruction.  
 See the PUNPCKHWD instruction.  
 See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PACKSSWB

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PACKSSWB mmreg1, mmreg2/mem64	0F 63h	Pack with saturation signed 16-bit operands into signed 8-bit results

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

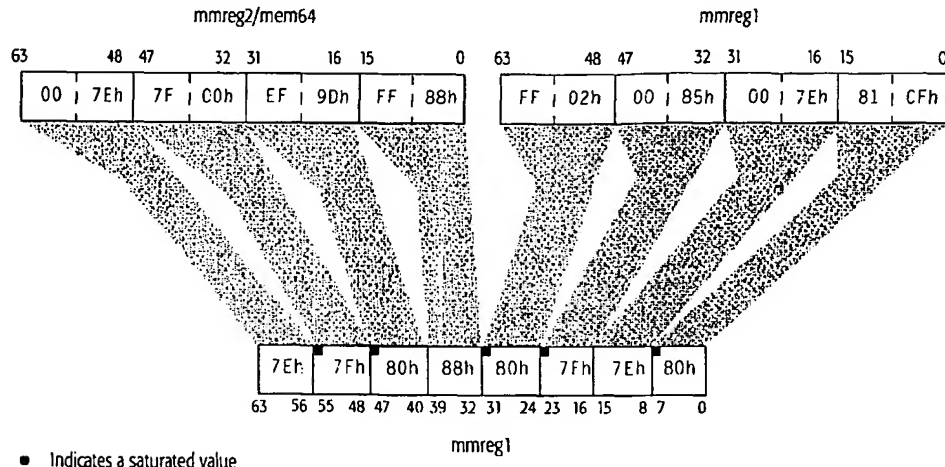
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PACKSSWB instruction performs a pack and saturate operation on four signed 16-bit values in the first operand and four signed 16-bit values in the second operand. The eight signed 8-bit results are placed in the specified MMX register.

The pack operation is a data conversion. The PACKSSWB instruction converts or packs the eight signed 16-bit values into eight signed 8-bit values, applying saturating arithmetic. If the signed 16-bit value is less than -128 (80h), it saturates to -128 (80h). If the signed 16-bit value is greater than 127 (7Fh), it saturates to 127 (7Fh). All values between -128 and 127 are represented by their signed 8-bit value.

The first operand must be an MMX register. In addition to providing the first operand, this MMX register is the location where the result of the pack and saturate operation is stored. The second operand can be an MMX register or a 64-bit memory location.

## Functional Illustration of the PACKSSWB Instruction



The following list explains the functional illustration of the PACKSSWB instruction:

- Bits 63–48 of the source operand (**mmreg2/mem64**) are packed into bits 63–56 of the destination operand (**mmreg1**). The result is not saturated because the 16-bit positive source operand (007Eh) does not exceed the capacity of a signed 8-bit destination operand.
- Bits 47–32 of the source operand are packed into bits 55–48 of the destination operand. The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand (7F00h) exceeds the capacity of a signed 8-bit destination operand.
- Bits 31–16 of the source operand are packed into bits 47–40 of the destination operand. The result is saturated to the largest possible 8-bit negative number because the 16-bit negative source operand (EF9Dh) exceeds the capacity of a signed 8-bit destination operand.
- Bits 15–0 of the source operand are packed into bits 39–32 of the destination operand. The result is not saturated because the 16-bit negative source operand (FF88h) does not exceed the capacity of the 8-bit destination operand.
- Bits 63–48 of the destination operand are packed into bits 31–24 of the destination operand. The result is saturated to the largest possible 8-bit negative number because the 16-bit negative source operand (FF02h) exceeds the capacity of a signed 8-bit destination operand.

---

## **A** *MMX Multimedia Technology*

- Bits 47–32 of the destination operand are packed into bits 23–16 of the destination operand. The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand (0085h) exceeds the capacity of a signed 8-bit destination operand.
- Bits 31–16 of the destination operand are packed into bits 15–8 of the destination operand. The result is not saturated because the 16-bit positive source operand (007Eh) does not exceed the capacity of a signed 8-bit destination operand.
- Bits 15–0 of the destination operand are packed into bits 7–0 of the destination operand. The result is saturated to the largest possible 8-bit negative number because the 16-bit negative source operand (81CFh) exceeds the capacity of a signed 8-bit destination operand.

**Related Instructions**      See the PACKSSDW instruction.  
                                    See the PACKUSWB instruction.  
                                    See the PUNPCKHBW instruction.  
                                    See the PUNPCKLBW instruction.

**PACKUSWB**

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PACKUSWB mmreg1, mmreg2/mem64	0F 67h	Pack with saturation signed 16-bit operands into unsigned 8-bit results

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

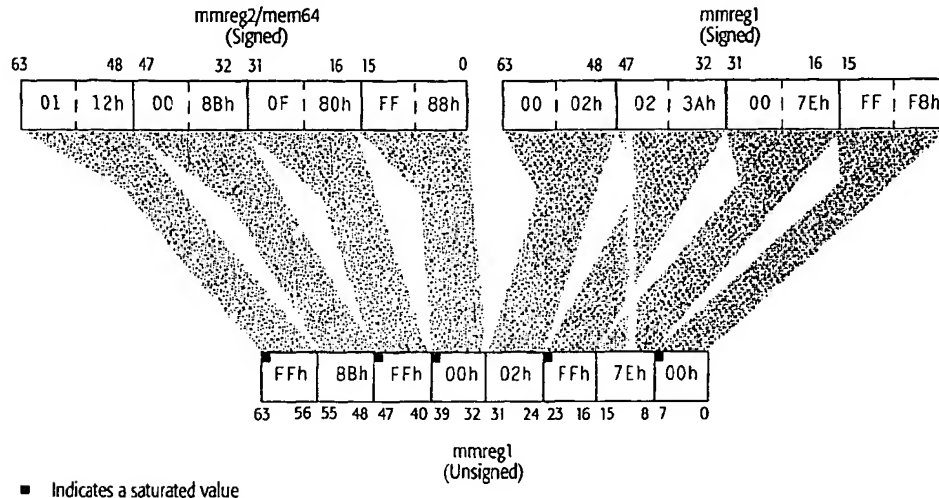
The PACKUSWB instruction performs a pack and saturate operation on four signed 16-bit values in the first operand and four signed 16-bit values in the second operand. The eight unsigned 8-bit results are placed in the specified MMX register.

The pack operation is a data conversion. The PACKUSWB instruction converts or packs the eight signed 16-bit values into eight unsigned 8-bit values, applying saturating arithmetic. If the signed 16-bit value is a negative number, it saturates to 0 (00h). If the signed 16-bit value is greater than 255 (FFh), it saturates to 255 (FFh). All values between 0 and 255 are represented with their unsigned 8-bit value.

The first operand must be an MMX register. In addition to providing the first operand, this MMX register is the location where the result of the pack and saturate operation is stored. The second operand can be an MMX register or a 64-bit memory location.

# A MMX Multimedia Technology

## Functional Illustration of the PACKUSWB Instruction



The following list explains the functional illustration of the PACKUSWB instruction:

- Bits 63–48 of the source operand (mmreg2/mem64) are packed into bits 63–56 of the destination operand (mmreg1). The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand (0112h) exceeds the capacity of an unsigned 8-bit destination operand.
- Bits 47–32 of the source operand are packed into bits 55–48 of the destination operand. The result is not saturated because the 16-bit positive source operand (008Bh) does not exceed the capacity of an unsigned 8-bit destination operand.
- Bits 31–16 of the source operand are packed into bits 47–40 of the destination operand. The result is saturated to the largest possible 8-bit positive number because the 16-bit positive source operand exceeds the capacity of an unsigned 8-bit destination operand.
- Bits 15–0 of the source operand are packed into bits 39–32 of the destination operand. The result is saturated to 00h because the source operand (FF88h) is a negative value.
- Bits 63–48 of the destination operand are packed into bits 31–24 of the destination operand (mmreg1). The result is not saturated because the 16-bit positive source operand (0002h) does not exceed the capacity of an unsigned 8-bit destination operand.
- Bits 47–32 of the destination operand are packed into bits 23–16 of the destination operand. The result is saturated to the largest possible 8-bit positive number

because the 16-bit positive source operand (023Ah) exceeds the capacity of an unsigned 8-bit destination operand.

- Bits 31–16 of the destination operand are packed into bits 15–8 of the destination operand. The result is not saturated because the 16-bit positive source operand (007Eh) does not exceed the capacity of an unsigned 8-bit destination operand.
- Bits 15–0 of the destination operand are packed into bits 7–0 of the destination operand. The result is saturated to 00h because the source operand (FFF8h) is a negative value.

**Related Instructions**      See the PACKSSDW instruction.  
                                    See the PACKSSWB instruction.  
                                    See the PUNPCKHBW instruction.  
                                    See the PUNPCKLBW instruction.



# A MMX Multimedia Technology

## PADDB

*mnemonic*                      *opcode*   *description*

PADDB mmreg1, mmreg2/mem64   0F FCh   Add unsigned packed 8-bit values

Privilege:                      none

Registers Affected:           MMX

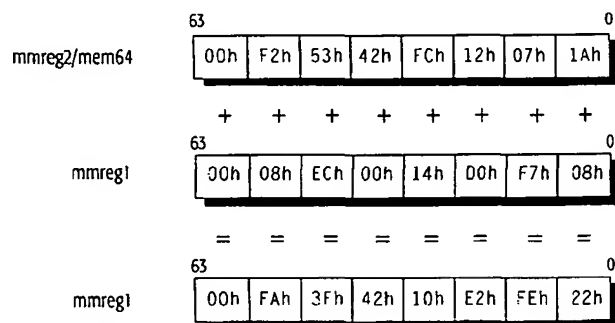
Flags Affected:               none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDB instruction adds eight unsigned 8-bit values from the source operand (an MMX register or a 64-bit memory location) to the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). If any of the eight results is greater than the capacity of its 8-bit destination, the value wraps around with no carry into the next location. The eight 8-bit results are stored in the MMX register that is specified as the destination operand.

## Functional Illustration of the PADDB Instruction



The following list explains the functional illustration of the PADDB instruction:

- The value 53h is added to ECh and wraps around to 3Fh.
- The value FCh is added to 14h and wraps around to 10h.
- The remaining addition operations are simple unsigned operations with no wraparound.

### Related Instructions

See the PADDD instruction.  
 See the PADDW instruction.  
 See the PADDSB instruction.  
 See the PADDSW instruction.  
 See the PADDUSB instruction.  
 See the PADDUSW instruction.

# A MMX Multimedia Technology

## PADDD

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PADDD mmreg1, mmreg2/mem64	0F FEh	Add unsigned packed 32-bit values

Privilege: none

Registers Affected: MMX

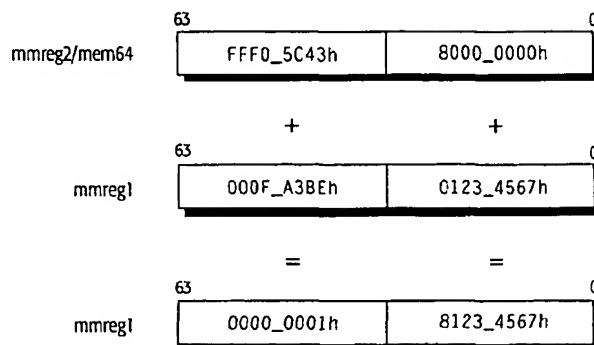
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDD instruction adds two unsigned 32-bit values from the source operand (an MMX register or a 64-bit memory location) to the two corresponding unsigned 32-bit values in the destination operand (an MMX register). If any of the two results is greater than the capacity of its 32-bit destination, the value wraps around with no carry into the next location. The two 32-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PADDD Instruction



The following list explains the functional illustration of the PADDD instruction:

- The value FFF0\_5C43h is added to 000F\_A3BEh and wraps around to 0000\_0001h.
- The second addition is a simple unsigned add operation with no wraparound.

**Related Instructions**      See the PADDB instruction.  
                                      See the PADDW instruction.  
                                      See the PADDSB instruction.  
                                      See the PADDSW instruction.

# A MMX Multimedia Technology

## PADDSB

*mnemonic*                      *opcode*   *description*

PADDSB mmreg1, mmreg2/mem64 OF ECh   Add signed packed 8-bit values and saturate

Privilege:                      none

Registers Affected:           MMX

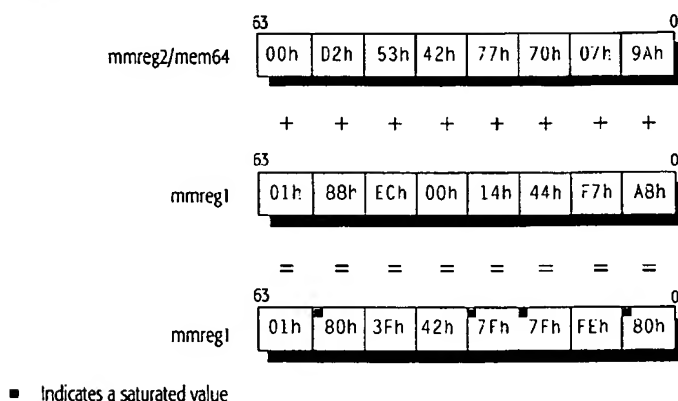
Flags Affected:                none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDSB instruction adds eight signed 8-bit values from the source operand (an MMX register or a 64-bit memory location) to the eight corresponding signed 8-bit values in the destination operand (an MMX register). If the sum of any two 8-bit values is less than -128 (80h), it saturates to -128 (80h). If the sum of any two 8-bit values is greater than 127 (7Fh), it saturates to 127 (7Fh). The eight signed 8-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PADDSB Instruction



The following list explains the functional illustration of the PADDSB instruction:

- The signed 8-bit positive value 00h is added to the signed 8-bit positive value 01h with a signed 8-bit positive result of 01h.
- The signed 8-bit negative value D2h (−46) is added to the signed 8-bit negative value 88h (−120) and saturates to 80h (−128), the largest possible signed 8-bit negative value.
- The signed 8-bit positive value 53h (+83) is added to the signed 8-bit negative value ECh (−20) with a signed 8-bit positive result of 3Fh (+63).
- The signed 8-bit positive value 42h is added to the signed 8-bit positive value 00h with a signed 8-bit positive result of 42h.
- The signed 8-bit positive value 77h (+119) is added to the signed 8-bit positive value 14h (+20) and saturates to 7Fh (+127), the largest possible positive value.
- The signed 8-bit positive value 70h (+112) is added to the signed 8-bit positive value 44h (+68) and saturates to 7Fh (+127), the largest possible positive value.
- The signed 8-bit positive value 07h (+7) is added to the signed 8-bit negative value F7h (−9) with a signed 8-bit negative result of FEh (−2).
- The signed 8-bit negative value 9Ah (−102) is added to the signed 8-bit negative value A8h (−88) and saturates to 80h (−128), the largest possible signed 8-bit negative value.

**Related Instructions**

- See the PADDB instruction.
- See the PADDD instruction.
- See the PADDW instruction.
- See the PADDSW instruction.

# A MMX Multimedia Technology

## PADDSW

*mnemonic*                      *opcode*   *description*

---

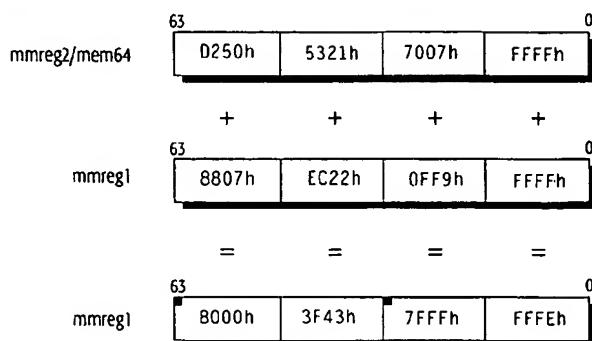
PADDSW mmreg1, mmreg2/mem64    0F EDh    Add signed packed 16-bit values and saturate

Privilege:                      none  
 Registers Affected:           MMX  
 Flags Affected:               none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDSW instruction adds four signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) to the four corresponding signed 16-bit values in the destination operand (an MMX register). If the sum of any two 16-bit values is less than -32768 (8000h), it saturates to -32768 (8000h). If the sum of any two 16-bit values is greater than 32767 (7FFFh), it saturates to 32767 (7FFFh). The four signed 16-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PADDSW Instruction



■ Indicates a saturated value

The following list explains the functional illustration of the PADDSW instruction:

- The signed 16-bit negative value D250h (–11696) is added to the signed 16-bit negative value 8807h (–30713) and saturates to 8000h (–32768), the largest possible signed 16-bit negative value.
- The signed 16-bit positive value 5321h (+21281) is added to the signed 16-bit negative value EC22h (–5086) with a signed 16-bit positive result of 3F43h (+16195).
- The signed 16-bit positive value 7007h (+28679) is added to the signed 16-bit positive value 0FF9h (+4089) and saturates to 7FFFh (+32767), the largest possible positive value.
- The signed 16-bit negative value FFFFh (–1) is added to the signed 16-bit negative value FFFFh (–1) with the negative 16-bit result of FFFEh (–2).

### Related Instructions

See the PADDB instruction.

See the PADDD instruction.

See the PADDW instruction.

See the PADDSB instruction.

See the PADDUSB instruction.

See the PADDUSW instruction.



# A MMX Multimedia Technology

## PADDUSB

*mnemonic* *opcode* *description*

PADDUSB mmreg1, mmreg2/mem64 0F DCh Add unsigned packed 8-bit values and saturate

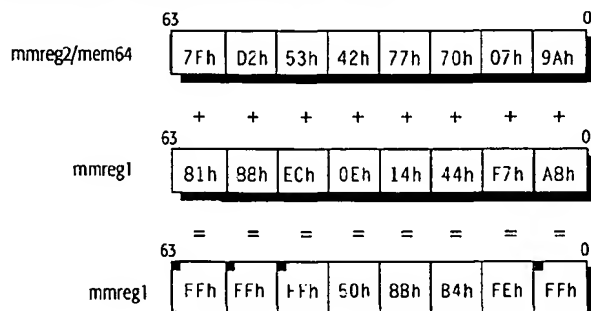
Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDUSB instruction adds eight unsigned 8-bit values from the source operand (an MMX register or a 64-bit memory location) to the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). The eight unsigned 8-bit results are stored in the MMX register specified as the destination operand.

If the sum of any two unsigned 8-bit values is greater than 255 (FFh), it saturates to 255 (FFh).

## Functional Illustration of the PADDUSB Instruction



■ Indicates a saturated value

The following list explains the functional illustration of the PADDUSB instruction:

- The sum of 7Fh and 81h is 100h. This value is greater than FFh, so the result saturates to FFh.
- The sum of D2h and 88h is 15Ah. This value is greater than FFh, so the result saturates to FFh.
- The sum of 53h and ECh is 13Fh. This value is greater than FFh, so the result saturates to FFh.
- The sum of 42h and 0Eh is 50h. This value is not greater than FFh, so the result does not saturate.
- The sum of 77h and 14h is 8Bh. This value is not greater than FFh, so the result does not saturate.
- The sum of 70h and 44h is B4h. This value is not greater than FFh, so the result does not saturate.
- The sum of 07h and F7h is FEh. This value is not greater than FFh, so the result does not saturate.
- The sum of 9Ah and A8h is 142h. This value is greater than FFh, so the result saturates to FFh.

### Related Instructions

See the PADDB instruction.

See the PADDD instruction.

See the PADDW instruction.

See the PADDSB instruction.

See the PADDSW instruction.

See the PADDUSW instruction.

# A MMX Multimedia Technology

## PADDUSW

*mnemonic* *opcode* *description*

PADDUSW mmreg1, mmreg2/mem64 0F DDh Add unsigned packed 16-bit values and saturate

Privilege: none

Registers Affected: MMX

Flags Affected: none

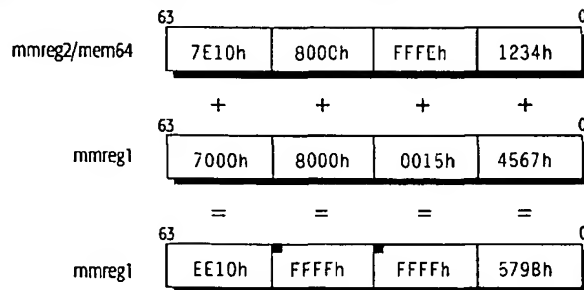
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDUSW instruction adds four unsigned 16-bit values from the source operand (an MMX register or a 64-bit memory location) to the four corresponding unsigned 16-bit values in the destination operand (an MMX register). The four unsigned 16-bit results are stored in the MMX register specified as the destination operand.

If the sum of any two unsigned 16-bit values is greater than 65,535 (FFFFh), it saturates to 65,535 (FFFFh).

## Functional Illustration of the PADDUSW Instruction



■ Indicates a saturated value

The following list explains the functional illustration of the PADDUSW instruction:

- The sum of 7E10h and 7000h is EE10h. This value is not greater than FFFFh, so the result does not saturate.
- The sum of 8000h and 8000h is 10000h. This value is greater than FFFFh, so the result saturates to FFFFh.
- The sum of FFEh and 0015h is 10013h. This value is greater than FFFFh, so the result saturates to FFFFh.
- The sum of 1234h and 4567h is 579Bh. This value is not greater than FFFFh, so the result does not saturate.

**Related Instructions**

- See the PADDB instruction.
- See the PADDD instruction.
- See the PADDW instruction.
- See the PADDSB instruction.
- See the PADDSW instruction.
- See the PADDUSB instruction.

# A MMX Multimedia Technology

## PADDW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PADDW mmreg1, mmreg2/mem64	0F FDh	Add unsigned packed 16-bit values

Privilege: none

Registers Affected: MMX

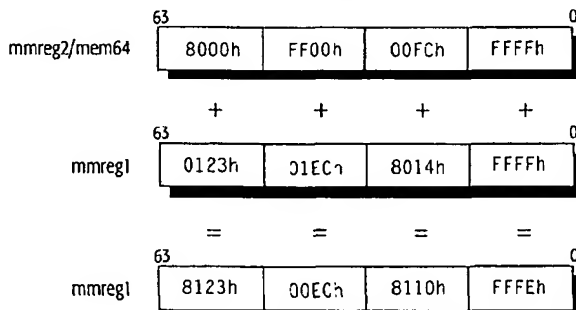
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PADDW instruction adds four unsigned 16-bit values from the source operand (an MMX register or a 64-bit memory location) to the four corresponding unsigned 16-bit values in the destination operand (an MMX register). If any of the four results is greater than the capacity of its 16-bit destination, the value wraps around with no carry into the next location. The four 16-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PADDW Instruction



The following list explains the functional illustration of the PADDW instruction:

- The value 8000h is added to 0123h with a normal unsigned result of 8123h.
- The value FF00h is added to 01EC7 and wraps around to 00ECh.
- The value 00FCh is added to 8014h with a normal signed result of 8110h.
- The value FFFFh is added to FFFFh and wraps around to FFFEh.

### Related Instructions

See the PADDB instruction.

See the PADDD instruction.

See the PADDSB instruction.

See the PADDSW instruction.

See the PADDUSB instruction.

See the PADDUSW instruction.

# A MMX Multimedia Technology

## PAND

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PAND mmreg1, mmreg2/mem64	0F DBh	AND 64-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

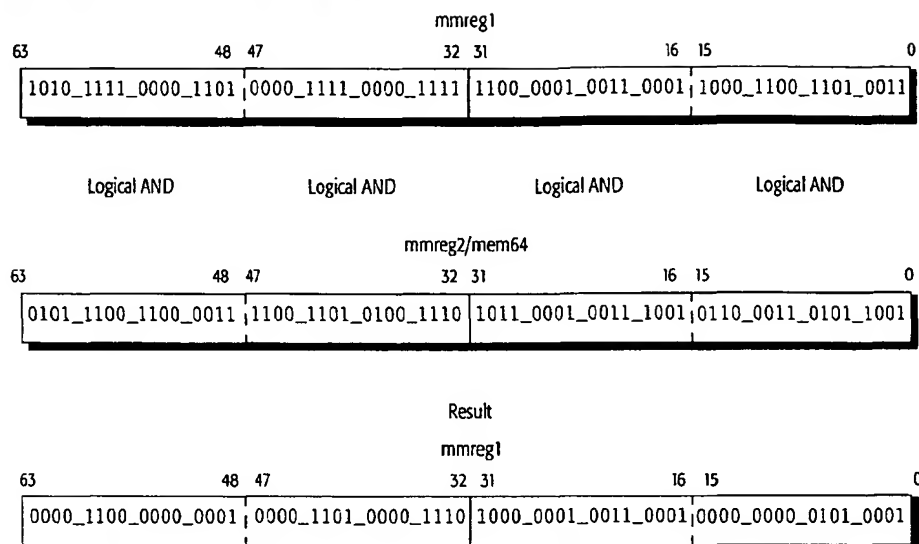
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PAND instruction operates on the 64-bit source and destination operands to complete a bitwise logical AND. The results are stored in the destination operand. If the corresponding bits in the source and destination operands both equal 1, the resulting bit is 1 in the destination. If either bit in the source or destination operands equals 0, the resulting bit is 0 in the destination.

The PAND instruction can be used to extract operands from packed fields based on the masks that are produced by the compare instructions—PCMPEQ and PCMPGT. This technique can eliminate branch prediction overhead in MMX routines.

## Functional Illustration of the PAND Instruction



### Related Instructions

See the PANDN instruction.

See the POR instruction.

See the PXOR instruction.



# A MMX Multimedia Technology

## PANDN

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PANDN mmreg1, mmreg2/mem64	OF DFh	Invert a 64-bit value, then AND the inverted value and a 64-bit value in memory or an MMX register

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

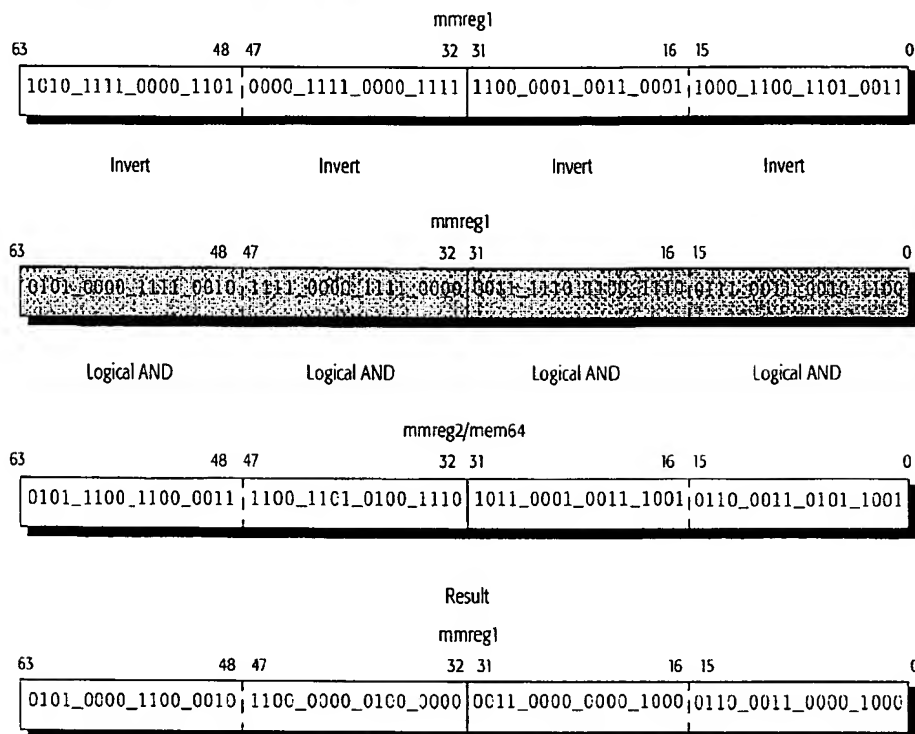
Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PANDN instruction first operates on the 64-bit destination operand (an MMX register) to complete a bitwise logical NOT, inverting each bit. This operation changes 1 bits to 0 bits and 0 bits to 1 bits, storing the results in the destination operand. The inverted 64-bit destination operand is then logically AND'd with the 64-bit source operand (an MMX register or a 64-bit memory operand) to complete the PANDN operation.

If corresponding bits in the source operand and the inverted destination operand are both 1, the resulting bit is 1 in the destination. If either bit in the source operand or the inverted destination operand is 0, the resulting bit is 0 in the destination.

The PANDN instruction can be used to extract alternate operands from packed fields based on the inverse of the masks that are produced by the compare instructions—PCMPEQ and PCMPGT. This technique can eliminate branch prediction overhead in MMX routines.

## Functional Illustration of the PANDN Instruction



### Related Instructions

See the PAND instruction.

See the POR instruction.

See the PXOR instruction.

# A MMX Multimedia Technology

## PCMPEQB

*mnemonic* *opcode* *description*

PCMPEQB mmreg1, mmreg2/mem64 0F 74h Compare packed 8-bit values for equality

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PCMPEQB instruction operates on 8-bit data values. The instruction compares two 8-bit values to determine if they are equal.

If the corresponding bits in the two operands are equal, all the bits in that 8 bits of the destination operand are set to 1. If any of the corresponding bits in the two operands are not equal, all the bits in that 8 bits of the destination operand are set to 0.

## Functional Illustration of the PCMPEQB Instruction

mmreg2/mem64	63	32				31	0	
	DBh	15h	43h	FFh	80h	CEh	A1h	04h
	Compare	Compare	Compare	Compare	Compare	Compare	Compare	Compare
mmreg1	63	32				31	0	
	DDh	15h	42h	FFh	80h	EEh	A1h	14h
	Result	Result	Result	Result	Result	Result	Result	Result
mmreg1	63	32				31	0	
	00h	FFh	00h	FFh	FFh	00h	FFh	00h
	False	True	False	True	True	False	True	False

## Related Instructions

See the PCMPEQD instruction.  
 See the PCMPEQW instruction.  
 See the PCMPGTB instruction.  
 See the PCMPGTD instruction.  
 See the PCMPGTW instruction.

# A MMX Multimedia Technology

## PCMPEQD

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PCMPEQD mmreg1, mmreg2/mem64	0F 76h	Compare packed 32-bit values for equality

Privilege: none

Registers Affected: MMX

Flags Affected: none

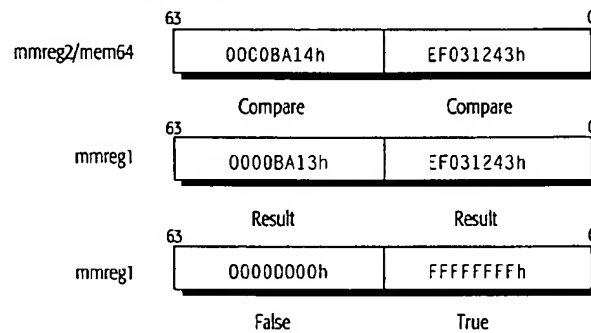
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PCMPEQD instruction operates on 32-bit data values. The instruction compares two 32-bit values to determine if they are equal.

If the corresponding bits in the two operands are equal, all the bits in that 32 bits of the destination operand are set to 1. If any of the corresponding bits in the two operands are not equal, all the bits in that 32 bits of the destination operand are set to 0.

## Functional Illustration of the PCMPEQD Instruction



## Related Instructions

- See the PCMPEQB instruction.
- See the PCMPEQW instruction.
- See the PCMPGTB instruction.
- See the PCMPGTD instruction.
- See the PCMPGTW instruction.

# A MMX Multimedia Technology

## PCMPEQW

*mnemonic* *opcode* *description*

PCMPEQW mmreg1, mmreg2/mem64 0F 75h Compare packed 16-bit values for equality

Privilege: none

Registers Affected: MMX

Flags Affected: none

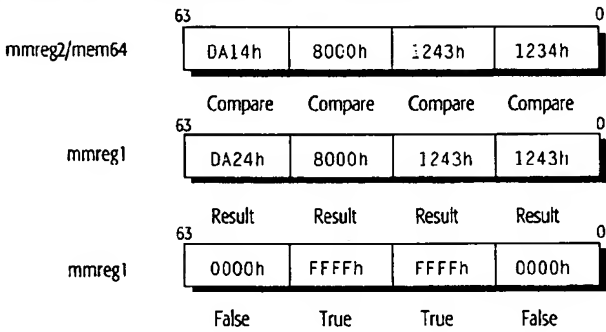
Exceptions Generated

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PCMPEQW instruction operates on 16-bit data values. The instruction compares two 16-bit values to determine if they are equal.

If the corresponding bits in the two operands are equal, all the bits in that 16 bits of the destination operand are set to 1. If any of the corresponding bits in the two operands are not equal, all the bits in that 16 bits of the destination operand are set to 0.

## Functional Illustration of the PCMPSEQW Instruction



**Related Instructions**

- See the PCMPSEQB instruction.
- See the PCMPSEQD instruction.
- See the PCMPGTB instruction.
- See the PCMPGTD instruction.
- See the PCMPGTW instruction.



# A MMX Multimedia Technology

## PCMPGTB

*mnemonic* *opcode* *description*

PCMPGTB mmreg1, mmreg2/mem64 0F 64h Compare signed packed 8-bit values for magnitude

Privilege: none

Registers Affected: MMX

Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PCMPGTB instruction operates on signed 8-bit data values. The instruction compares two signed 8-bit values to determine if the value in the destination operand is greater than the corresponding signed 8-bit data value in the source operand.

If the value in the destination operand is greater than the value in the source operand, all the bits in that 8 bits of the destination operand are set to 1. If the value in the destination operand is equal to or less than the value in the source operand, all the bits in that 8 bits of the destination operand are set to 0.

## Functional Illustration of the PCMPGTB Instruction

mmreg2/mem64	63	32				31	0	
	DCh	25h	41h	FFh	80h	7Fh	A6h	04h
	Greater?	Greater?	Greater?	Greater?	Greater?	Greater?	Greater?	Greater?
mmreg1	63	32				31	0	
	DDh	24h	42h	01h	80h	80h	A3h	14h
	Result	Result	Result	Result	Result	Result	Result	Result
mmreg1	63	32				31	0	
	FFh	00h	FFh	FFh	00h	00h	00h	FFh
	True	False	True	True	False	False	False	True

The following list explains the functional illustration of the PCMPGTB instruction:

- The negative value DDh (–35) is greater than the negative value DCh (–36), so the result is true (FFh).
- The positive value 24h (+36) is not greater than the positive value 25h (+37), so the result is false (00h).
- The positive value 42h (+66) is greater than the positive value 41h (+65), so the result is true (FFh).
- The positive value 01h (+1) is greater than the negative value FFh (–1), so the result is true (FFh).
- The negative value 80h (–128) is not greater than the negative value 80h (–128), so the result is false (00h).
- The negative value 80h (–128) is not greater than the positive value 7Fh (+127), so the result is false (00h).
- The negative value A3h (–93) is not greater than the negative value A6h (–90), so the result is false (00h).
- The positive value 14h (+20) is greater than the positive value 04h (+4), so the result is true (FFh).

### Related Instructions

See the PCMPEQB instruction.  
 See the PCMPEQD instruction.  
 See the PCMPEQW instruction.  
 See the PCMPGTD instruction.  
 See the PCMPGTW instruction.

# A MMX Multimedia Technology

## PCMPGTD

*mnemonic* *opcode* *description*

PCMPGTD mmreg1, mmreg2/mem64 0F 66h Compare signed packed 32-bit values for magnitude

Privilege: none

Registers Affected: MMX

Flags Affected: none

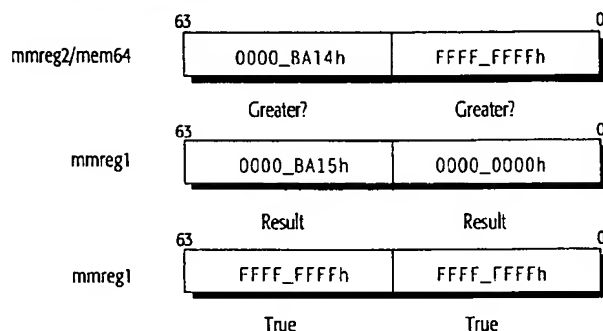
Exceptions Generated

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PCMPGTD instruction operates on signed 32-bit data values. The instruction compares two signed 32-bit values to determine if the value in the destination operand is greater than the corresponding signed 32-bit data value in the source operand.

If the value in the destination operand is greater than the value in the source operand, all the bits in that 32 bits of the destination operand are set to 1. If the value in the destination operand is equal to or less than the value in the source operand, all the bits in that 32 bits of the destination operand are set to 0.

## Functional Illustration of the PCMPGTD Instruction



The following list explains the functional illustration of the PCMPGTD instruction:

- The positive value `0000_BA15h` (+47637) is greater than the positive value `0000_BA14h` (+47636), so the result is true (`FFFF_FFFFh`).
- The positive value `0000_0001h` (+1) is greater than the negative value `FFFF_FFFFh` (-1), so the result is true (`FFFF_FFFFh`).

### Related Instructions

See the PCMPQEB instruction.  
 See the PCMPQEQD instruction.  
 See the PCMPQEQW instruction.  
 See the PCMPGTB instruction.  
 See the PCMPGTW instruction.

# A MMX Multimedia Technology

## PCMPGTW

*mnemonic* *opcode* *description*

PCMPGTW mmreg1, mmreg2/mem64 0F 65h Compare signed packed 16-bit values for magnitude

Privilege: none

Registers Affected: MMX

Flags Affected: none

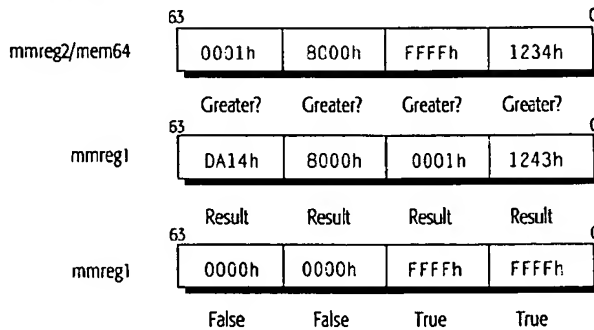
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PCMPGTW instruction operates on signed 16-bit data values. The instruction compares two signed 16-bit values to determine if the value in the destination operand is greater than the corresponding signed 16-bit data value in the source operand.

If the value in the destination operand is greater than the value in the source operand, all the bits in that 16 bits of the destination operand are set to 1. If the value in the destination operand is equal to or less than the value in the source operand, all the bits in that 16 bits of the destination operand are set to 0.

## Functional Illustration of the PCMPGTW Instruction



The following list explains the functional illustration of the PCMPGTW instruction:

- The negative value DA14h (–9708) is not greater than the positive value 0001h (+1), so the result is false (0000h).
- The negative value 8000h (–32768) is not greater than the negative value 8000h (–32768), so the result is false (0000h).
- The positive value 0001h (+1) is greater than the negative value FFFFh (–1), so the result is true (FFFFh).
- The positive value 1243h (+4675) is greater than the positive value 1234h (+4660), so the result is true (FFFFh).

### Related Instructions

See the PCMPEQB instruction.  
 See the PCMPEQD instruction.  
 See the PCMPEQW instruction.  
 See the PCMPGTB instruction.  
 See the PCMPGTD instruction.

# A MMX Multimedia Technology

## PMADDWD

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PMADDWD mmreg1, mmreg2/mem64	0F 5h	Multiply signed packed 16-bit values and add the 32-bit results

Privilege: none

Registers Affected: MMX

Flags Affected: none

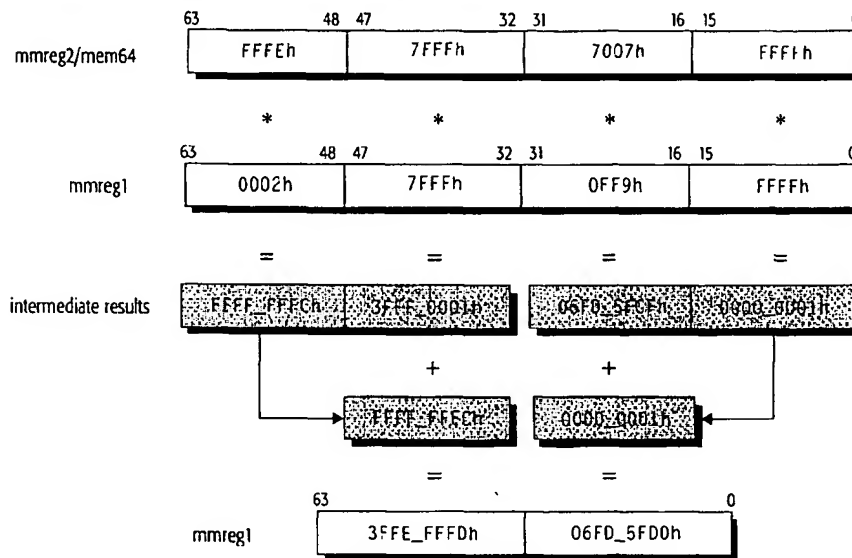
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PMADDWD instruction multiplies signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) by the corresponding signed 16-bit values in the destination operand (an MMX register), adds the resulting 32-bit values from the left and right halves of the 64-bit work space, and stores the 32-bit sums in the MMX destination register.

*Note: If all four of the 16-bit operands are 8000h, the result wraps around to 8000\_0000h because the maximum negative 16-bit value of 8000h multiplied by itself equals 4000\_0000h, and 4000\_0000h added to 4000\_0000h equals 8000\_0000h. The result of multiplying two negative numbers should be a positive number, but 8000\_0000h is the maximum possible 32-bit negative number rather than a positive number. This is the only instance of wraparound that can occur as a result of the PMADDWD instruction.*

## Functional Illustration of the PMADDWD Instruction



The following list explains the functional illustration of the PMADDWD instruction:

- The signed 16-bit negative value `FFFEh` ( $-2$ ) is multiplied by the signed 16-bit positive value `0002h` to produce a signed 32-bit negative intermediate result of `FFFF_FFFCh` ( $-4$ ).
- The signed 16-bit positive value `7FFFh` is multiplied by the signed 16-bit positive value `7FFFh` to produce a signed 32-bit positive intermediate result of `3FFF_0001h`.
- The two 32-bit intermediate results are added together to produce the final signed 32-bit positive result of `3FFE_FFFDh`.
- The signed 16-bit positive value `7007h` is multiplied by the signed 16-bit positive value `0FF9h` to produce a signed 32-bit intermediate result of `06FD_5FCFh`.
- The signed 16-bit negative value `FFFFh` ( $-1$ ) is multiplied by the signed 16-bit negative value `FFFFh` ( $-1$ ) to produce a signed 32-bit positive intermediate result of `0000_0001h`.
- The two 32-bit intermediate results are added together to produce the final signed 32-bit positive result of `06FD_5FD0h`.

**Related Instructions**      See the PMULHW instruction.  
                                      See the PMULLW instruction.



# A MMX Multimedia Technology

## PMULHW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PMULHW mmreg1, mmreg2/mem64	0F E5h	Multiply signed packed 16-bit values and store the high 16 bits

Privilege: none

Registers Affected: MMX

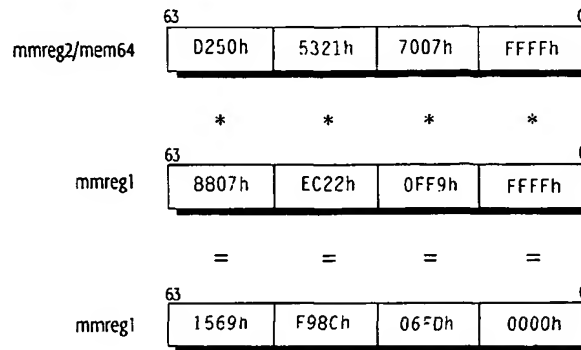
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PMULHW instruction multiplies four signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) by the four corresponding signed 16-bit values in the destination operand (an MMX register) and then stores the high-order 16 bits of the result (including the sign bit) in the destination operand.

## Functional Illustration of the PMULHW Instruction



The following list explains the functional illustration of the PMULHW instruction:

- The signed 16-bit negative value D250h (–2DB0h) is multiplied by the signed 16-bit negative value 8807h (–77F9h) to produce the signed 32-bit positive result of 1569\_4030h. The signed high-order 16-bits of the result are stored in the destination operand.
- The signed 16-bit positive value 5321h is multiplied by the signed 16-bit negative value EC22h (–13DEh) to produce the signed 32-bit negative result of F98C\_7662h (–0673\_899Eh). The signed high-order 16-bits of the result are stored in the destination operand.
- The signed 16-bit positive value 7007h is multiplied by the signed 16-bit positive value 0FF9h to produce the signed 32-bit positive result of 06FD\_5FCFh. The signed high-order 16-bits of the result are stored in the destination operand.
- The signed 16-bit negative value FFFFh (–1) is multiplied by the signed 16-bit negative value FFFFh (–1) to produce the signed 32-bit positive result of 0000\_0001h. The signed high-order 16-bits of the result are stored in the destination operand.

**Related Instructions**      See the PMADDWD instruction.  
                                      See the PMULLW instruction.  
                                      See the PUNPCKHWD instruction.  
                                      See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PMULLW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PMULLW mmreg1, mmreg2/mem64	0F D5h	Multiply signed packed 16-bit values and store the low 16 bits

Privilege: none

Registers Affected: MMX

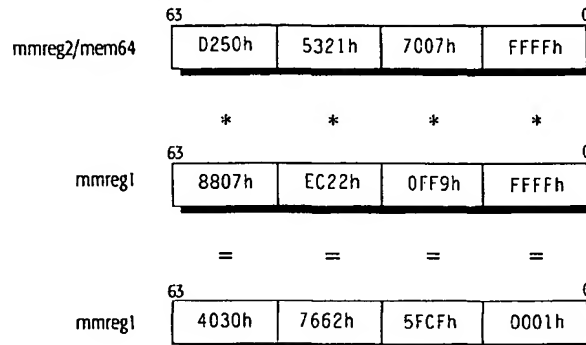
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PMULLW instruction multiplies four signed 16-bit values from the source operand (an MMX register or a 64-bit memory location) by the four corresponding signed 16-bit values in the destination operand (an MMX register) and then stores the low-order 16 bits of the result (unsigned) in the destination operand.

## Functional Illustration of the PMULLW Instruction



The following list explains the functional illustration of the PMULLW instruction:

- The signed 16-bit negative value D250h (–2DB0h) is multiplied by the signed 16-bit negative value 8807h (–77F9h) to produce the signed 32-bit positive result of 1569\_4030h. The unsigned low-order 16-bits of the result are stored in the destination operand.
- The signed 16-bit positive value 5321h is multiplied by the signed 16-bit negative value EC22h (–13DEh) to produce the signed 32-bit negative result of F98C\_7662h (–0673\_899Eh). The unsigned low-order 16-bits of the result are stored in the destination operand.
- The signed 16-bit positive value 7007h is multiplied by the signed 16-bit positive value 0FF9h to produce the signed 32-bit positive result of 06FD\_5FCFh. The unsigned low-order 16-bits of the result are stored in the destination operand.
- The signed 16-bit negative value FFFFh (–1) is multiplied by the signed 16-bit negative value FFFFh (–1) to produce the signed 32-bit positive result of 0000\_0001h. The unsigned low-order 16-bits of the result are stored in the destination operand.

### Related Instructions

See the PMADDWD instruction.

See the PMULHW instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## POR

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
POR mmreg1, mmreg2/mem64	0F EBh	OR 64-bit values

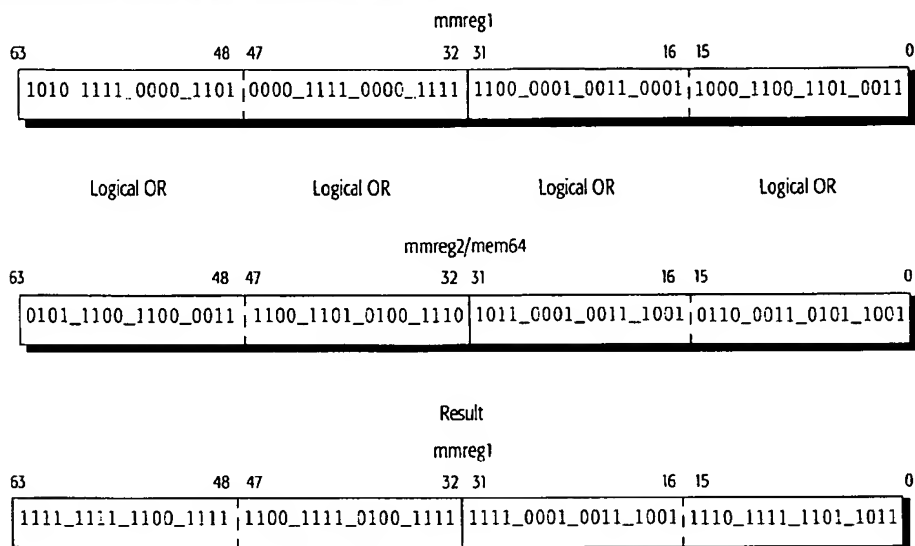
Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The POR instruction logically ORs the 64 bits of the source operand (an MMX register or a 64-bit memory location) with the 64 bits of the destination operand (an MMX register) and stores the result in the destination register.

A logical OR produces a 1 bit if either or both input bits is a 1. If both input bits are 0, a logical OR produces a 0 bit.

## Functional Illustration of the POR Instruction



In the functional illustration of the POR instruction, the 64-bit source value is logically OR'd to the 64-bit destination value, and the result is stored in the destination register.

**Related Instructions**

- See the PAND instruction.
- See the PANDN instruction.
- See the PXOR instruction.

# A MMX Multimedia Technology

## PSLLD

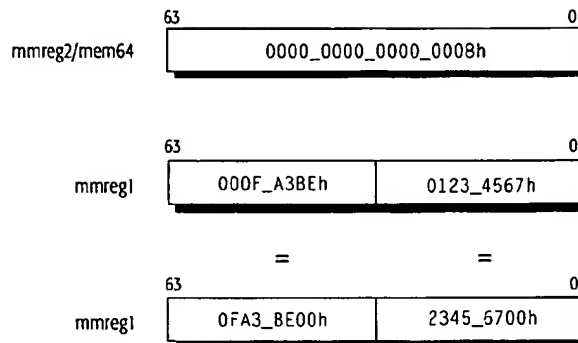
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSLLD mmreg1, mmreg2/mem64	0F F2h	Shift left logical packed 32-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the right
PSLLD mmreg1, imm8	0F 72h /6	Shift left logical packed 32-bit values in mmreg1 the number of positions in imm8 with zero fill from the right

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSLLD instruction shifts the two 32-bit operands in the destination operand (an MMX register) to the left by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the right. The two 32-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSLLD Instruction



The following list explains the functional illustration of the PSLLD instruction:

- The value 0000\_0000\_0000\_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the left.
- The 32-bit value 000F\_A3BEh in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 0FA3\_BE00h.
- The 32-bit value 0123\_4567h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 2345\_6700h.

### Related Instructions

See the PSLLQ instruction.  
 See the PSLLW instruction.  
 See the PSRAD instruction.  
 See the PSRAW instruction.  
 See the PSRLD instruction.  
 See the PSRLQ instruction.  
 See the PSRLW instruction.



# A MMX Multimedia Technology

## PSLLQ

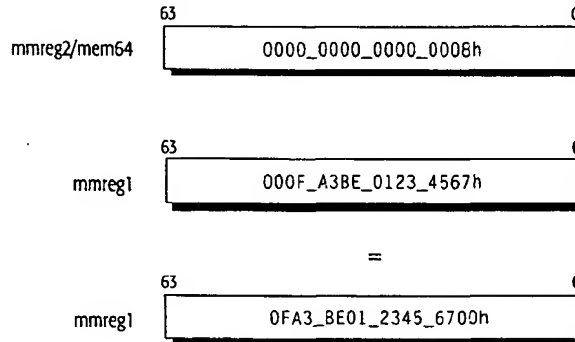
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSLLQ mmreg1, mmreg2/mem64	0F F3h	Shift left logical 64-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the right
PSLLQ mmreg1, imm8	0F 73h /6	Shift left logical 64-bit values in mmreg1 the number of positions in imm8 with zero fill from the right

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSLLQ instruction shifts the 64-bit operand in the destination operand (an MMX register) to the left by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted value is zero filled from the right. The 64-bit result is stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSLLQ Instruction



The following list explains the functional illustration of the PSLLQ instruction:

- The value 0000\_0000\_0000\_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the left.
- The 64-bit value 000F\_A3BE\_0123\_4567h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 0FA3\_BE01\_2345\_6700h.

**Related Instructions**

- See the PSLLD instruction.
- See the PSLLW instruction.
- See the PSRAD instruction.
- See the PSRAW instruction.
- See the PSRLD instruction.
- See the PSRLQ instruction.
- See the PSRLW instruction.

# A MMX Multimedia Technology

## PSLLW

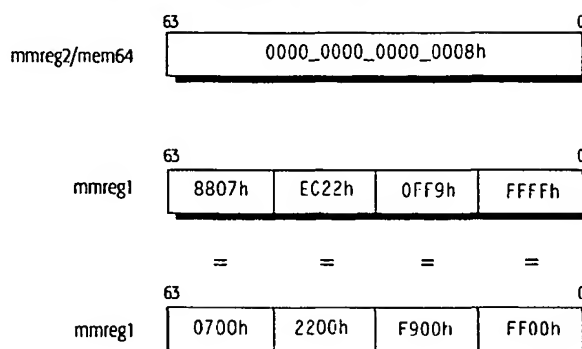
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSLLW mmreg1, mmreg2/mem64	0F F1h	Shift left logical packed 16-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the right
PSLLW mmreg1, imm8	0F 71h /6	Shift left logical packed 16-bit values in mmreg1 the number of positions in imm8 with zero fill from the right

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSLLW instruction shifts the four 16-bit operands in the destination operand (an MMX register) to the left by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the right. The four 16-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSLW Instruction



The following list explains the functional illustration of the PSLW instruction:

- The value 0000\_0000\_0000\_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the left.
- The 16-bit value 8807h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 0700h.
- The 16-bit value EC22h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as 2200h.
- The 16-bit value 0FF9h in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as F900h.
- The 16-bit value FFFFh in mmreg1 is shifted 8 bit positions to the left and stored in mmreg1 as FF00h.

### Related Instructions

See the PSLLD instruction.  
See the PSLLQ instruction.  
See the PSRAD instruction.  
See the PSRAW instruction.  
See the PSRLD instruction.  
See the PSRLQ instruction.  
See the PSRLW instruction.

# A MMX Multimedia Technology

## PSRAD

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSRAD mmreg1, mmreg2/mem64	0F E2h	Shift right arithmetic packed signed 32-bit values in mmreg1 the number of positions in mmreg2/mem64 with sign fill from the left
PSRAD mmreg1, imm8	0F 72h /4	Shift right arithmetic packed signed 32-bit values in mmreg1 the number of positions in imm8 with sign fill from the left

Privilege: none

Registers Affected: MMX

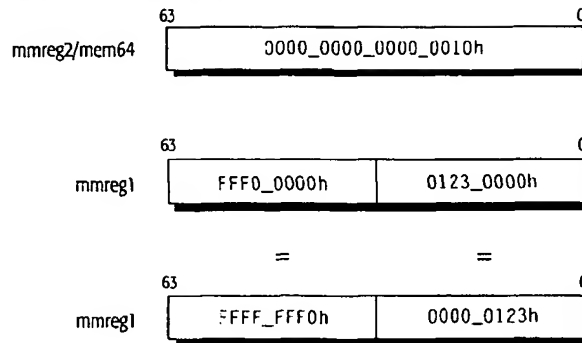
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSRAD instruction shifts the two signed 32-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are sign filled from the left. The two signed 32-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSRAD Instruction



The following list explains the functional illustration of the PSRAD instruction:

- The value 0000\_0000\_0000\_0010h in mmreg2/mem64 indicates a shift of 16 bit positions to the right.
- The 32-bit negative value FFF0\_0000h in mmreg1 is shifted 16 bit positions to the right with sign fill from the left and stored in mmreg1 as FFFF\_FFF0h.
- The 32-bit positive value 0123\_0000h in mmreg1 is shifted 16 bit positions to the right with sign fill from the left and stored in mmreg1 as 0000\_0123h.

### Related Instructions

See the PSLLD instruction.  
See the PSLLQ instruction.  
See the PSLLW instruction.  
See the PSRAW instruction.  
See the PSRLD instruction.  
See the PSRLQ instruction.  
See the PSRLW instruction.  
See the PUNPCKHWD instruction.  
See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PSRAW

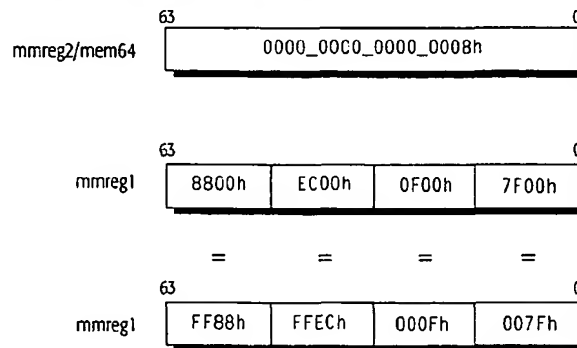
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSRAW mmreg1, mmreg2/mem64	0F E1h	Shift right arithmetic packed signed 16-bit values in mmreg1 the number of positions in mmreg2/mem64 with sign fill from the left
PSRAW mmreg1, imm8	0F 71h /4	Shift right arithmetic packed signed 16-bit values in mmreg1 the number of positions in imm8 with sign fill from the left

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSRAW instruction shifts the four signed 16-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are sign filled from the left. The four signed 16-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSRAW Instruction



The following list explains the functional illustration of the PSRAW instruction:

- The value 0000\_0000\_0000\_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the right.
- The 16-bit negative value 8800h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as FF88h.
- The 16-bit negative value EC00h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as FFECh.
- The 16-bit positive value 0F00h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as 000Fh.
- The 16-bit positive value 7F00h in mmreg1 is shifted 8 bit positions to the right with sign fill from the left and stored in mmreg1 as 007Fh.

**Related Instructions**

- See the PSLLD instruction.
- See the PSLLQ instruction.
- See the PSLLW instruction.
- See the PSRAD instruction.
- See the PSRLD instruction.
- See the PSRLQ instruction.
- See the PSRLW instruction.
- See the PUNPCKHBW instruction.
- See the PUNPCKLBW instruction.



# A MMX Multimedia Technology

## PSRLD

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSRLD mmreg1, mmreg2/mem64	0F D2h	Shift right logical packed 32-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the left
PSRLD mmreg1, imm8	0F 72h /2	Shift right logical packed 32-bit values in mmreg1 the number of positions in imm8 with zero fill from the left

Privilege: none

Registers Affected: MMX

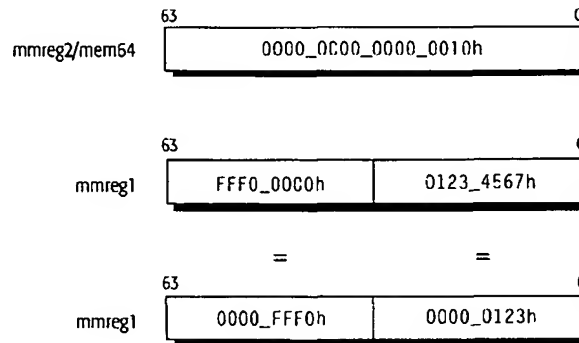
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSRLD instruction shifts the two 32-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the left. The two 32-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSRLD Instruction



The following list explains the functional illustration of the PSRLD instruction:

- The value 0000\_0000\_0000\_0010h in mmreg2/mem64 indicates a shift of 16 bit positions to the right.
- The 32-bit value FFF0\_0000h in mmreg1 is shifted 16 bit positions to the right and stored in mmreg1 as 0000\_FFF0h
- The 32-bit value 0123\_4567h in mmreg1 is shifted 16 bit positions to the right and stored in mmreg1 as 0000\_0123h.

**Related Instructions** See the PSLLD instruction.  
See the PSLLQ instruction.  
See the PSLLW instruction.  
See the PSRAD instruction.  
See the PSRAW instruction.  
See the PSRLQ instruction.  
See the PSRLW instruction.

# A MMX Multimedia Technology

## PSRLQ

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSRLQ mmreg1, mmreg2/mem64	0F D3h	Shift right logical 64-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the left
PSRLQ mmreg1, imm8	0F 73h /2	Shift right logical 64-bit values in mmreg1 the number of positions in imm8 with zero fill from the left

Privilege: none

Registers Affected: MMX

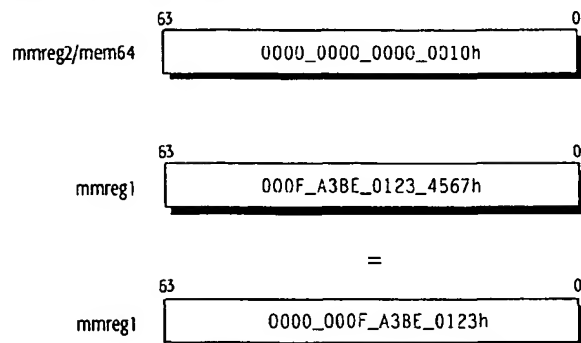
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSRLQ instruction shifts the 64-bit operand in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted value is zero filled from the left. The result is stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSRLQ Instruction



The following list explains the functional illustration of the PSRLQ instruction:

- The value 0000\_0000\_0000\_0010h in mmreg2/mem64 indicates a shift of 16 bit positions to the right.
- The 64-bit value 000F\_A3BE\_0123\_4567h in mmreg1 is shifted 16 bit positions to the right and stored in mmreg1 as 0000\_000F\_A3BE\_0123h.

**Related Instructions**

- See the PSLLD instruction.
- See the PSLLQ instruction.
- See the PSLLW instruction.
- See the PSRAD instruction.
- See the PSRAW instruction.
- See the PSRLD instruction.
- See the PSRLW instruction.

# A MMX Multimedia Technology

## PSRLW

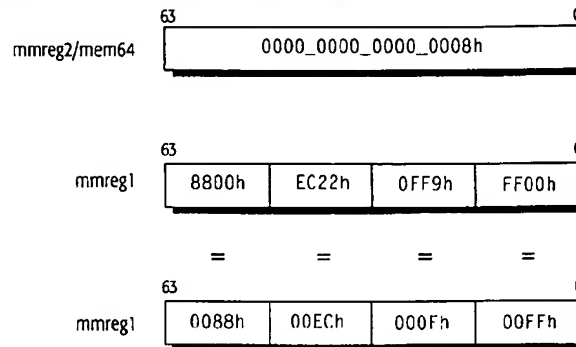
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSRLW mmreg1, mmreg2/mem64	0F D1h	Shift right logical packed 16-bit values in mmreg1 the number of positions in mmreg2/mem64 with zero fill from the left
PSRLW mmreg1, imm8	0F 71h /2	Shift right logical packed 16-bit values in mmreg1 the number of positions in imm8 with zero fill from the left

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSRLW instruction shifts the four 16-bit operands in the destination operand (an MMX register) to the right by the number of bit positions indicated by mmreg2/mem64 or by imm8, the 8-bit immediate operand. The shifted values are zero filled from the left. The four 16-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSRLW Instruction



The following list explains the functional illustration of the PSRLW instruction:

- The value 0000\_0000\_0000\_0008h in mmreg2/mem64 indicates a shift of 8 bit positions to the right.
- The 16-bit value 8800h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 0088h.
- The 16-bit value EC22h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 00ECh.
- The 16-bit value 0FF9h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 000Fh.
- The 16-bit value FF00h in mmreg1 is shifted 8 bit positions to the right and stored in mmreg1 as 00FFh.

### Related Instructions

See the PSLLD instruction.

See the PSLLQ instruction.

See the PSLLW instruction.

See the PSRAD instruction.

See the PSRAW instruction.

See the PSRLD instruction.

See the PSRLQ instruction.

# A MMX Multimedia Technology

## PSUBB

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSUBB mmreg1, mmreg2/mem64	0F FBh	Subtract unsigned packed 8-bit values with wraparound

Privilege: none

Registers Affected: MMX

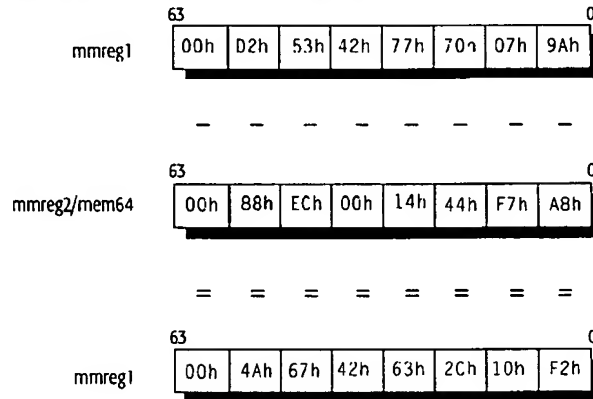
Flags Affected: none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBB instruction subtracts eight unsigned 8-bit values in the source operand (an MMX register or a 64-bit memory location) from the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). If the source operand is larger than the destination operand, the result wraps around.

## Functional Illustration of the PSUBB Instruction



The following list explains the functional illustration of the PSUBB instruction:

- The unsigned 8-bit value ECh is subtracted from the unsigned 8-bit value 53h and wraps around to 67h.
- The unsigned 8-bit value F7h is subtracted from the unsigned 8-bit value 07h and wraps around to 10h.
- The unsigned 8-bit value A8h is subtracted from the unsigned 8-bit value 9Ah and wraps around to F2h.
- All the remaining operations are simple subtraction with no wraparound.

**Related Instructions**

- See the PSUBD instruction.
- See the PSUBW instruction.
- See the PSUBSB instruction.
- See the PSUBSW instruction.
- See the PSUBUSB instruction.
- See the PSUBUSW instruction.



# A MMX Multimedia Technology

## PSUBD

*mnemonic*                      *opcode*   *description*

PSUBD mmreg1, mmreg2/mem64   0F FAh   Subtract unsigned packed 32-bit values with wraparound

Privilege:                      none

Registers Affected:           MMX

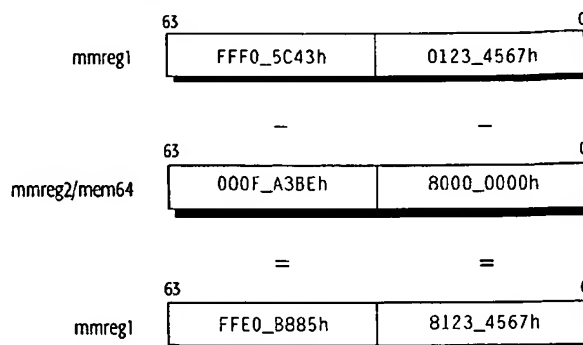
Flags Affected:               none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBD instruction subtracts two unsigned 32-bit values in the source operand (an MMX register or a 64-bit memory location) from the two corresponding unsigned 32-bit values in the destination operand (an MMX register). If the source operand is larger than the destination operand, the result wraps around.

## Functional Illustration of the PSUBD Instruction



The following list explains the functional illustration of the PSUBD instruction:

- The unsigned 32-bit value 8000\_0000h is subtracted from the unsigned 32-bit value 0123\_4567h and wraps around to 8123\_4567h.
- The remaining operation is a simple subtraction with no wraparound.

### Related Instructions

See the PSUBB instruction.  
 See the PSUBW instruction.  
 See the PSUBSB instruction.  
 See the PSUBSW instruction.  
 See the PSUBUSB instruction.  
 See the PSUBUSW instruction.

# A MMX Multimedia Technology

## PSUBSB

*mnemonic*                      *opcode*    *description*

---

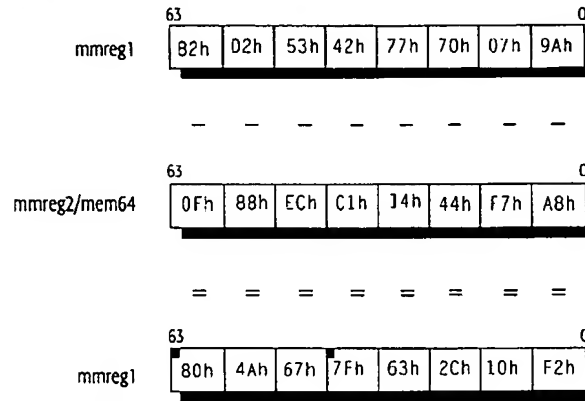
PSUBSB mmreg1, mmreg2/mem64 0F E8h    Subtract signed packed 8-bit values and saturate

Privilege:                      none  
 Registers Affected:           MMX  
 Flags Affected:               none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBSB instruction subtracts eight signed 8-bit values in the source operand (an MMX register or a 64-bit memory location) from the eight corresponding signed 8-bit values in the destination operand (an MMX register). If a result is less than -128 (80h), it saturates to -128 (80h). If a result is greater than 127 (7Fh), it saturates to 127 (7Fh). The eight signed 8-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSUBSB Instruction



The following list explains the functional illustration of the PSUBSB instruction:

- The signed 8-bit positive value 0Fh is subtracted from the signed 8-bit negative value 82h, and the result saturates to 80h because it is less than 80h, the smallest possible signed 8-bit value.
- The signed 8-bit negative value C1h is subtracted from the signed 8-bit positive value 42h, and the result saturates to 7Fh because it is greater than 7Fh, the largest possible signed 8-bit value.
- All the remaining operations are simple signed subtraction with no saturation.

**Related Instructions**

- See the PSUBB instruction.
- See the PSUBD instruction.
- See the PSUBW instruction.
- See the PSUBSW instruction.
- See the PSUBUSB instruction.
- See the PSUBUSW instruction.

# A MMX Multimedia Technology

## PSUBSW

*mnemonic*                      *opcode*    *description*

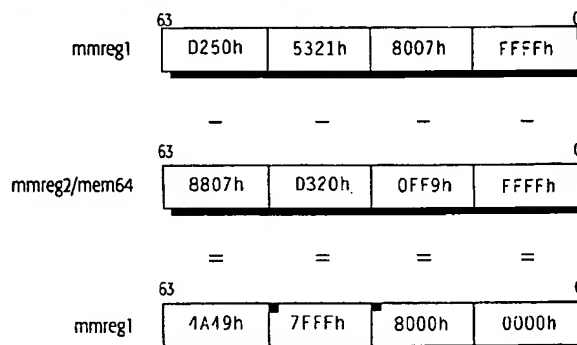
---

PSUBSW mmreg1, mmreg2/mem64 0F E9h    Subtract signed packed 16-bit values and saturate

Privilege:                      none  
 Registers Affected:           MMX  
 Flags Affected:               none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBSW instruction subtracts four signed 16-bit values in the source operand (an MMX register or a 64-bit memory location) from the four corresponding signed 16-bit values in the destination operand (an MMX register). If a result is less than -32768 (8000h), it saturates to -32768 (8000h). If a result is greater than 32767 (7FFFh), it saturates to 32767 (7FFFh). The four signed 16-bit results are stored in the MMX register specified as the destination operand.

**Functional Illustration of the PSUBSW Instruction**

- Indicates a saturated value

The following list explains the functional illustration of the PSUBSW instruction:

- The signed 16-bit negative value D320h is subtracted from the signed 16-bit positive value 5321h, and the result saturates to 7FFFh because it is greater than 7FFFh, the largest possible signed 16-bit value.
- The signed 16-bit positive value 0FF9h is subtracted from the signed 16-bit negative value 8007h, and the result saturates to 8000h because it is less than 8000h, the smallest possible signed 16-bit value.
- The remaining operations are simple signed subtraction with no saturation.

**Related Instructions**

See the PSUBB instruction.  
 See the PSUBD instruction.  
 See the PSUBW instruction.  
 See the PSUBSB instruction.  
 See the PSUBUSB instruction.  
 See the PSUBUSW instruction.

# A MMX Multimedia Technology

## PSUBUSB

*mnemonic*                      *opcode*   *description*

PSUBUSB mmreg1, mmreg2/mem64    0F D8h   Subtract unsigned packed 8-bit values and saturate

Privilege:                      none

Registers Affected:           MMX

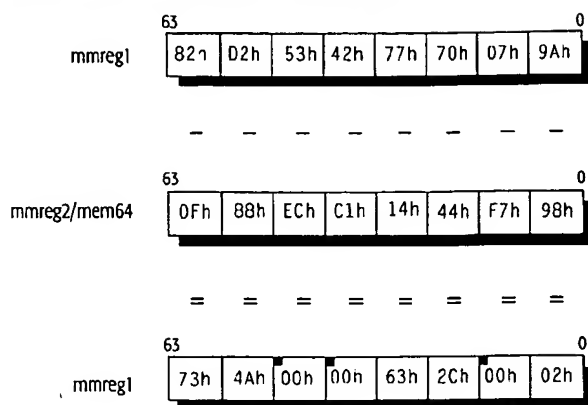
Flags Affected:               none

Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBUSB instruction subtracts eight unsigned 8-bit values in the source operand (an MMX register or a 64-bit memory location) from the eight corresponding unsigned 8-bit values in the destination operand (an MMX register). If any 8-bit source value is greater than its corresponding 8-bit destination value, the result saturates to 00h. The eight unsigned 8-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSUBUSB Instruction



■ Indicates a saturated value

The following list explains the functional illustration of the PSUBUSB instruction:

- The unsigned 8-bit value ECh is subtracted from the unsigned 8-bit value 53h, and the result saturates to 00h because the source operand is greater than the destination operand.
- The unsigned 8-bit value C1h is subtracted from the unsigned 8-bit value 42h, and the result saturates to 00h because the source operand is greater than the destination operand.
- The unsigned 8-bit value F7h is subtracted from the unsigned 8-bit value 07h, and the result saturates to 00h because the source operand is greater than the destination operand.
- All the remaining operations are simple unsigned subtraction with no saturation.

### Related Instructions

See the PSUBB instruction.  
See the PSUBD instruction.  
See the PSUBW instruction.  
See the PSUBSB instruction.  
See the PSUBSW instruction.  
See the PSUBUSW instruction.



# A MMX Multimedia Technology

## PSUBUSW

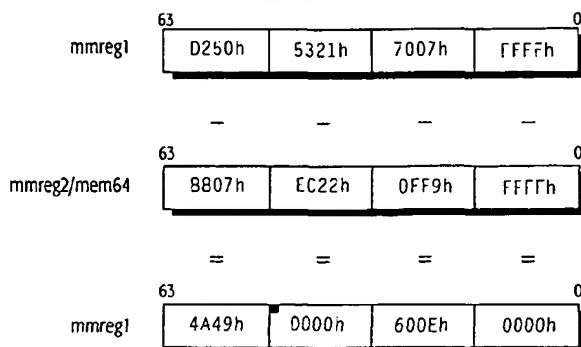
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSUBUSW mmreg1, mmreg2/mem64	0F D9h	Subtract unsigned packed 16-bit values and saturate

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBUSW instruction subtracts four unsigned 16-bit values in the source operand (an MMX register or a 64-bit memory location) from the four corresponding unsigned 16-bit values in the destination operand (an MMX register). If any 16-bit source value is greater than its corresponding 16-bit destination value, the result saturates to 0000h. The four unsigned 16-bit results are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PSUBUSW Instruction



- Indicates a saturated value

The following list explains the functional illustration of the PSUBUSW instruction:

- The unsigned 16-bit value EC22h is subtracted from the unsigned 16-bit value 5321h, and the result saturates to 0000h because the source operand is greater than the destination operand.
- The remaining operations are simple unsigned subtraction with no saturation.

### Related Instructions

See the PSUBB instruction.  
 See the PSUBD instruction.  
 See the PSUBW instruction.  
 See the PSUBSB instruction.  
 See the PSUBSW instruction.  
 See the PSUBUSB instruction.

# A MMX Multimedia Technology

## PSUBW

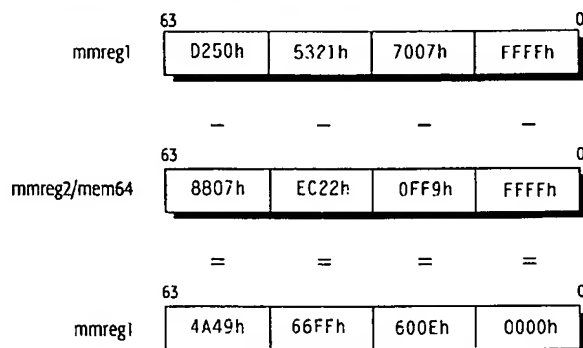
<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PSUBW mmreg1, mmreg2/mem64	0F F9h	Subtract unsigned packed 16-bit values with wraparound

Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PSUBW instruction subtracts four unsigned 16-bit values in the source operand (an MMX register or a 64-bit memory location) from the four corresponding unsigned 16-bit values in the destination operand (an MMX register). If the source operand is larger than the destination operand, the result wraps around.

## Functional Illustration of the PSUBW Instruction



The following list explains the functional illustration of the PSUBW instruction:

- The unsigned 16-bit value EC22h is subtracted from the unsigned 16-bit value 5321h and the result wraps around to 66FFh.
- The remaining operations are simple unsigned subtraction with no saturation.

**Related Instructions**

- See the PSUBB instruction.
- See the PSUBD instruction.
- See the PSUBSB instruction.
- See the PSUBSW instruction.
- See the PSUBUSB instruction.
- See the PSUBUSW instruction.

# A MMX Multimedia Technology

## PUNPCKHBW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PUNPCKHBW mmreg1, mmreg2/mem64	0F 68h	Unpack the high 32 bits of packed 8-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

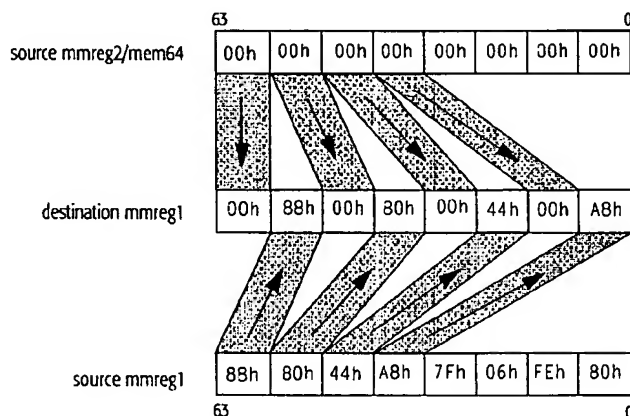
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PUNPCKHBW instruction unpacks and interleaves four 8-bit values from the high 32 bits of the source operand (an MMX register or a 64-bit memory location) and four 8-bit values from the high 32 bits of the destination operand (an MMX register). The 8-bit values from the source operand become the high 8 bits of the 16-bit results, and the 8-bit values from the destination operand become the low 8 bits of the 16-bit results. The eight interleaved 8-bit values are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PUNPCKHBW Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKHBW instruction, the 8-bit values from mmreg1 are stored in the low-order 8 bits of the 16-bit result. The mmreg2/mem64 source operand is set to all zero bits so it can provide zero fill in the high-order 8 bits of the 16-bit result. This is a method that can be used to expand unsigned 8-bit values into unsigned 16-bit operands for subsequent processing that requires higher precision.

### Related Instructions

See the PACKSSWB instruction.  
 See the PACKUSWB instruction.  
 See the PSRAW instruction.  
 See the PUNPCKHDQ instruction.  
 See the PUNPCKHWD instruction.  
 See the PUNPCKLBW instruction.  
 See the PUNPCKLDQ instruction.  
 See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PUNPCKHDQ

*mnemonic* *opcode* *description*

PUNPCKHDQ mmreg1, mmreg2/mem64 0F 6Ah Unpack the high 32 bits of packed 32-bit values

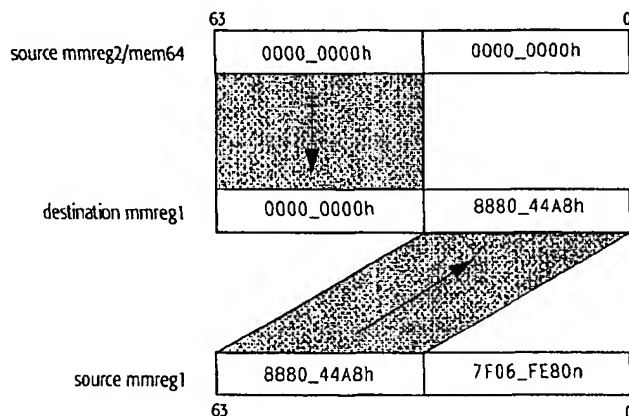
Privilege: none  
 Registers Affected: MMX  
 Flags Affected: none  
 Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PUNPCKHDQ instruction unpacks and interleaves the high 32 bits of the source operand (an MMX register or a 64-bit memory location) and the high 32 bits of the destination operand (an MMX register). The 32-bit value from the source operand becomes the high 32 bits of the 64-bit result, and the 32-bit value from the destination operand becomes the low 32 bits of the 64-bit result. The interleaved 32-bit values are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PUNPCKHDQ Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKHDQ instruction, the 32-bit value from mmreg1 is stored in the low-order 32 bits of the 64-bit result. The mmreg2/mem64 source operand is set to all zero bits so it can provide zero fill in the high-order 32 bits of the 64-bit result. This is a method that can be used to expand unsigned 32-bit values into unsigned 64-bit operands for subsequent processing that requires higher precision.

### Related Instructions

See the PUNPCKHBW instruction.

See the PUNPCKHWD instruction.

See the PUNPCKLBW instruction.

See the PUNPCKLDQ instruction.

See the PUNPCKLWD instruction.



# A MMX Multimedia Technology

## PUNPCKHWD

*mnemonic* *opcode* *description*

PUNPCKHWD mmreg1, mmreg2/mem64 0F 69h Unpack the high 32 bits of packed 16-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

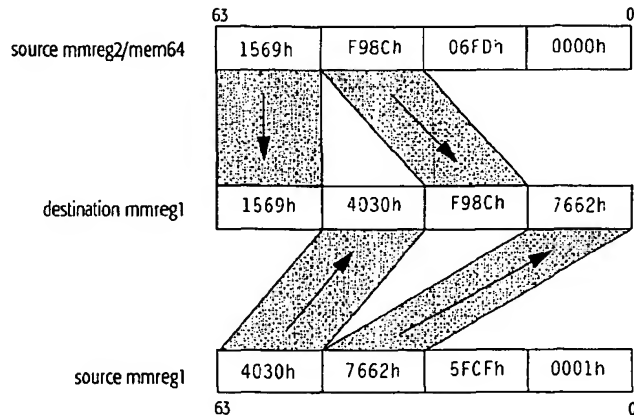
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PUNPCKHWD instruction unpacks and interleaves two 16-bit values from the high 32 bits of the source operand (an MMX register or a 64-bit memory location) and two 16-bit values from the high 32 bits of the destination operand (an MMX register). The 16-bit values from the source operand become the high 16 bits of the 32-bit results, and the 16-bit values from the destination operand become the low 16 bits of the 32-bit results. The four interleaved 16-bit values are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PUNPCKHWD Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKHWD instruction, the 16-bit values from mmreg1 are stored in the low-order 16 bits of the 32-bit result. The 16-bit values from the mmreg2/mem64 source operand are stored in the high-order 16 bits of the 32-bit result. This is an example of the use of the PUNPCKHWD instruction to assemble 32-bit operands from the high and low 16-bit results produced by the PMULHW and PMULLW instructions. In this example, the high and low 16-bit results are interleaved to produce the signed 32-bit results 1569\_4030h and F98C\_7662h.

### Related Instructions

- See the PACKSSDW instruction.
- See the PSRAD instruction.
- See the PMULHW instruction.
- See the PMULLW instruction.
- See the PUNPCKHBW instruction.
- See the PUNPCKHDQ instruction.
- See the PUNPCKLBW instruction.
- See the PUNPCKLDQ instruction.
- See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PUNPCKLBW

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PUNPCKLBW mmreg1, mmreg2/mem64	OF 60h	Unpack the low 32-bits of packed 8-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

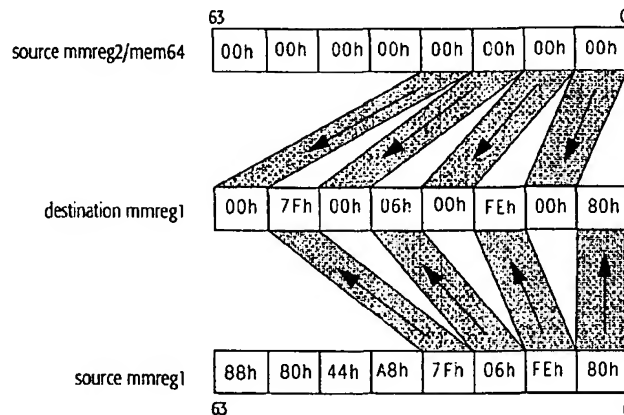
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PUNPCKLBW instruction unpacks and interleaves four 8-bit values from the low 32 bits of the source operand (an MMX register or a 64-bit memory location) and four 8-bit values from the low 32 bits of the destination operand (an MMX register). The 8-bit values from the source operand become the high 8 bits of the 16-bit results, and the 8-bit values from the destination operand become the low 8 bits of the 16-bit results. The eight interleaved 8-bit values are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PUNPCKLBW Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKLBW instruction, the 8-bit values from mmreg1 are stored in the low-order 8 bits of the 16-bit result. The mmreg2/mem64 source operand is set to all zero bits so it can provide zero fill in the high-order 8 bits of the 16-bit result. This is a method that can be used to expand unsigned 8-bit values into unsigned 16-bit operands for subsequent processing that requires higher precision.

### Related Instructions

- See the PACKSSWB instruction.
- See the PACKUSWB instruction.
- See the PSRAW instruction.
- See the PUNPCKHBW instruction.
- See the PUNPCKHDQ instruction.
- See the PUNPCKHWD instruction.
- See the PUNPCKLDQ instruction.
- See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PUNPCKLDQ

*mnemonic* *opcode* *description*

PUNPCKLDQ mmreg1, mmreg2/mem64 0F 62h Unpack the low 32 bits of packed 32-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

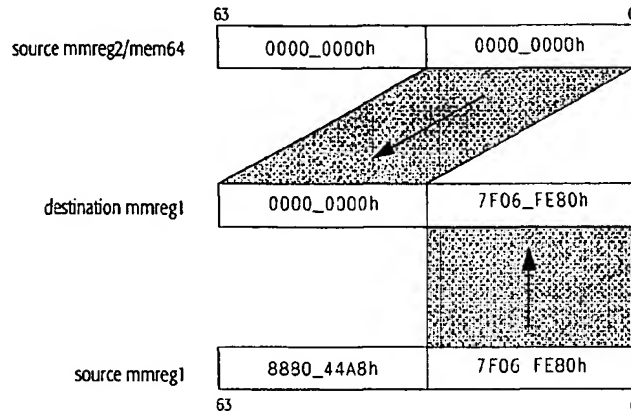
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PUNPCKLDQ instruction unpacks and interleaves the low 32 bits of the source operand (an MMX register or a 64-bit memory location) and the low 32 bits of the destination operand (an MMX register). The 32-bit value from the source operand becomes the high 32 bits of the 64-bit result, and the 32-bit value from the destination operand becomes the low 32 bits of the 64-bit result. The interleaved 32-bit values are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PUNPCKLDQ Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKLDQ instruction, the 32-bit value from mmreg1 is stored in the low-order 32 bits of the 64-bit result. The mmreg2/mem64 source operand is set to all zero bits so it can provide zero fill in the high-order 32 bits of the 64-bit result. This is a method that can be used to expand unsigned 32-bit values into unsigned 64-bit operands for subsequent processing that requires higher precision.

### Related Instructions

- See the PUNPCKHBW instruction.
- See the PUNPCKHDQ instruction.
- See the PUNPCKHWD instruction.
- See the PUNPCKLBW instruction.
- See the PUNPCKLWD instruction.

# A MMX Multimedia Technology

## PUNPCKLWD

*mnemonic* *opcode* *description*

PUNPCKLWD mmreg1, mmreg2/mem64 OF 61h Unpack the low 32 bits of packed 16-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

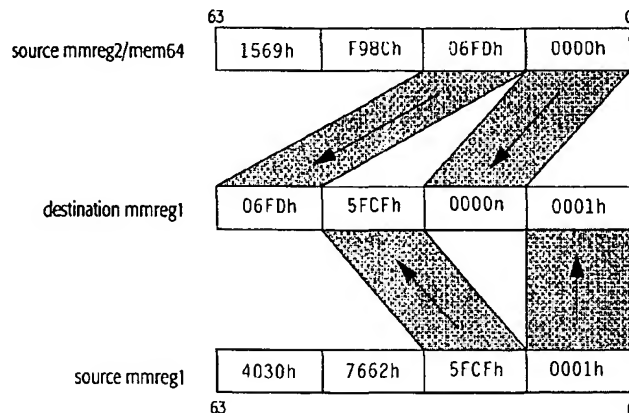
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PUNPCKLWD instruction unpacks and interleaves two 16-bit values from the low 32 bits of the source operand (an MMX register or a 64-bit memory location) and two 16-bit values from the low 32 bits of the destination operand (an MMX register). The 16-bit values from the source operand become the high 16 bits of the 32-bit results, and the 16-bit values from the destination operand become the low 16 bits of the 32-bit results. The four interleaved 16-bit values are stored in the MMX register specified as the destination operand.

## Functional Illustration of the PUNPCKLWD Instruction

In the following figure, the destination register is shown at the center to illustrate the flow of data from the two source operands.



In the functional illustration of the PUNPCKLWD instruction, the 16-bit values from mmreg1 are stored in the low-order 16 bits of the 32-bit result. The 16-bit values from the mmreg2/mem64 source operand are stored in the high-order 16 bits of the 32-bit result. This is an example of the use of the PUNPCKLWD instruction to assemble 32-bit operands from the high and low 16-bit results produced by the PMULHW and PMULLW instructions. In this example, the high and low 16-bit results are interleaved to produce the signed 32-bit results 06FD\_5FCFh and 0000\_0001h.

### Related Instructions

- See the PACKSSWD instruction.
- See the PSRAD instruction.
- See the PMULHW instruction.
- See the PMULLW instruction.
- See the PUNPCKHBW instruction.
- See the PUNPCKHDQ instruction.
- See the PUNPCKHWD instruction.
- See the PUNPCKLBW instruction.
- See the PUNPCKLDQ instruction.



# A MMX Multimedia Technology

## PXOR

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
PXOR mmreg1, mmreg2/mem64	0F EFh	XOR 64-bit values

Privilege: none

Registers Affected: MMX

Flags Affected: none

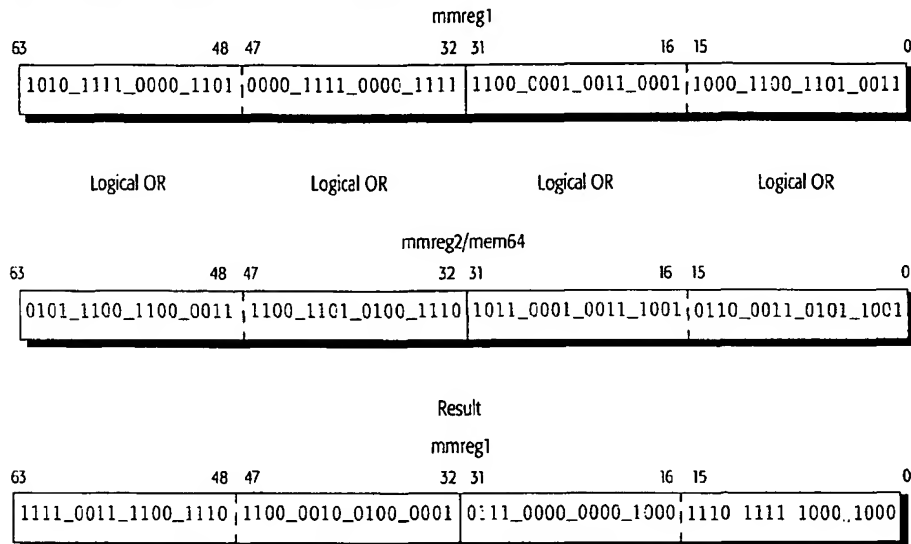
Exceptions Generated:

Exception	Real	Virtual 8086	Protected	Description
Invalid opcode (6)	X	X	X	The emulate MMX instruction bit (EM) of the control register (CR0) is set to 1.
Device not available (7)	X	X	X	Save the floating-point or MMX state if the task switch bit (TS) of the control register (CR0) is set to 1.
Stack exception (12)			X	During instruction execution, the stack segment limit was exceeded.
General protection (13)			X	During instruction execution, the effective address of one of the segment registers used for the operand points to an illegal memory location.
Segment overrun (13)	X	X		One of the instruction data operands falls outside the address range 00000h to 0FFFFh.
Page fault (14)		X	X	A page fault resulted from the execution of the instruction.
Floating-point exception pending (16)	X	X	X	An exception is pending due to the floating-point execution unit.
Alignment check (17)		X	X	An unaligned memory reference resulted from the instruction execution, and the alignment mask bit (AM) of the control register (CR0) is set to 1. (In Protected Mode, CPL = 3.)

The PXOR instruction logically XORs the 64 bits of the source operand (an MMX register or a 64-bit memory location) with the 64 bits of the destination operand (an MMX register) and stores the result in the destination register.

A logical XOR produces a 1 bit if only one of the two input bits is a 1. If both input bits are 0 or both input bits are 1, a logical XOR produces a 0 bit.

## Functional Illustration of the PXOR Instruction



In the functional illustration of the PXOR instruction, the 64-bit source value is logically XOR'd to the 64-bit destination value, and the result is stored in the destination register.

**Related Instructions**

- See the PAND instruction.
- See the PANDN instruction.
- See the POR instruction.

---

# **A** *MMX Multimedia Technology*

---

# ***Appendix B***

## **Code Optimization**

### **Introduction**

---

The AMD-K6 3D processor can efficiently execute code written for previous-generation x86 processors. However, to get the highest performance from the unique microarchitecture of the processor, certain code optimization techniques should be applied.

This appendix contains information to assist programmers in creating optimized code for the processor. This information is targeted at compiler/assembler designers and assembly language programmers writing high-performance code sequences. It is assumed that the reader possesses an in-depth knowledge of the x86 architecture.

## **The AMD-K6 Family of Processors**

---

Processors in the AMD-K6 family use a decoupled instruction decode and superscalar execution microarchitecture, including state-of-the-art RISC design techniques, to deliver sixth-generation performance with full x86 binary software compatibility. An x86 binary-compatible processor implements the industry-standard x86 instruction set by decoding and executing the x86 instruction set as its native mode of operation. Only this native mode permits delivery of maximum performance when running PC software.

## **The AMD-K6 3D Processor**

---

The AMD-K6 3D processor brings superscalar RISC performance to desktop systems running industry-standard x86 software. This processor implements advanced design techniques such as:

- Instruction pre-decoding
- Multiple x86 opcode decoding
- Single-cycle internal RISC operations
- Multiple parallel execution units
- Out-of-order execution
- Data-forwarding
- Register renaming
- Dynamic branch prediction

The processor is capable of issuing, executing, and retiring multiple x86 instructions per cycle, resulting in superior scaleable performance.

Although the processor is capable of extracting code parallelism out of off-the-shelf, commercially available x86 software, specific code optimizations for the AMD-K6 3D processor can result in significantly higher delivered performance. This appendix describes the RISC86 microarchitecture in the processor and makes recommendations for optimizing execution of x86 software on

the processor. The coding techniques for achieving peak performance on the AMD-K6 3D processor include, but are not limited to, those recommended for the Pentium, Pentium II, and Pentium Pro processors. However, many of these optimizations are not necessary for the AMD-K6 3D processor to achieve maximum performance. For example, due to more flexible pipeline control in the microarchitecture, the AMD-K6 3D processor is less sensitive to instruction selection and the scheduling of code. This flexibility is one of the distinct advantages of the AMD-K6 3D processor microarchitecture.

In addition to the ability to perform multimedia operations, the AMD-K6 3D processor includes the first implementation of the 3D instruction set. 3D technology was created based on suggestions from leading graphics and software vendors. Utilizing a data format and Single Instruction Multiple Data (SIMD) operations based on the MMX instruction model, the processor can produce up to four, 32-bit, single-precision floating-point results per clock cycle. 3D technology also includes new integer multimedia instructions, a new instruction to allow the prefetching of data under software control, and a faster enter/exit multimedia-state instruction. For more information, see Chapter 4, “3D Technology” on page 81 and Appendix A, “MMX Multimedia Technology” on page 347.

The 3D units provide support for high-performance, floating-point vector operations, which can replace x87 instructions and enhance the performance of 3D graphics and other floating-point-intensive applications. The complete multimedia processing unit in the processor combines existing MMX instructions with the new 3D instructions. The 3D instructions share the use of the MMX registers with the multimedia unit. By merging 3D instructions with MMX instructions, it now becomes possible to write x86 programs containing both integer and floating-point instructions without a performance penalty for intermixing MMX and x87 floating-point instructions. All these improvements have been carefully designed to bring a better multimedia experience to mainstream PC users while maintaining backwards compatibility with all existing x86 software.

## **Execution Units and Dependency Latencies**

---

The processor contains several specialized execution pipelines—store, load, register X, register Y, floating-point, and branch condition. Each pipeline operates independently and handles a specific subset of the RISC86 instruction set. The register X and register Y pipelines each contain integer, multimedia, and 3D execution resources, some of which are shared between the two. This section describes the operation of these units, their execution latencies, and how these latencies affect concurrent dependency chains.

*Note: The multimedia execution unit executes MMX instructions.*

A dependency occurs when data needed in one execution unit/resource is being processed in another unit/resource (or a different stage of the same unit/resource). Additional latencies can occur because the dependent execution unit must wait for the data from the supplying unit. Table 86 on page 465 provides a summary of the execution units, the operations performed within these units, the operation latency, and the operation throughput.

### **Execution Unit Terminology**

<b>Introduction</b>	The execution units operate with two different types of register values—operands and results. Of these there are three types of operands and two types of results.
<b>Operands</b>	<p>The three types of operands are as follows:</p> <ul style="list-style-type: none"><li>■ <i>Address register operands</i>—used for address calculations of load and store operations</li><li>■ <i>Data register operands</i>—used for register operations</li><li>■ <i>Store data register operands</i>—used for memory stores</li></ul>
<b>Results</b>	<p>The two types of results are as follows:</p> <ul style="list-style-type: none"><li>■ <i>Data register results</i>—produced by load or register operations</li><li>■ <i>Address register results</i>—produced by Lea or Push operations</li></ul>

The following examples illustrate the operand and result definitions:

Add AX, BX

The Add operation has two data register operands (AX, and BX) and one data register result (AX).

Load BX, [SP+4·CX+8]

The Load operation has two address register operands (SP and CX as base and index registers, respectively) and a data register result (BX).

Store [SP+4·CX+8], AX

The Store operation has a store data register operand (AX) and two address register operands (SP and CX as base and index registers, respectively).

Lea SI, [SP+4·CX+8]

The Lea operation (a type of store operation) has address register operands (SP and CX as base and index registers, respectively), and an address register result.

## Six-Stage Pipeline

To help visualize the operations within the AMD-K6 3D processor, Figure 115 illustrates the effective pipeline stages. This is a simplified illustration in that the processor contains multiple parallel pipelines (starting after common instruction fetch and x86 decode pipe stages), and these pipelines often execute operations out-of-order with respect to each other. This view of the processor execution pipeline illustrates the effect of execution latencies for various types of operations.

For many instructions, the effective pipeline is seven stages. For register operations that do not require execution stage 2, the effective pipeline is six stages.

Instruction Fetch	x86→RISC86 Decode	RISC86 Op Issue	Operand Fetch	Execution Stage 1	Execution Stage 2*	Commit
----------------------	----------------------	--------------------	------------------	----------------------	-----------------------	--------

**Note:** \* Execution Stage 2 is optional

**Figure 115. Processor Pipeline**



## **B** *Code Optimization*

### **Register Execution Units**

The register execution resources are attached to the register X unit execution pipeline and the register Y unit execution pipeline. Each register execution pipeline has dedicated resources that consist of an integer execution unit and an multimedia/ALU execution unit. In addition, both pipelines can use shared execution units for 3D operations and MMX shift and multiply operations. Figure 6 on page 20 shows the details of the register X and Y execution pipelines.

The register X integer ALU execution resource can execute all ALU operations including ALU, multiply, divide (signed and unsigned), shift, and rotate. Data register results are available after a minimum of one clock of execution latency.

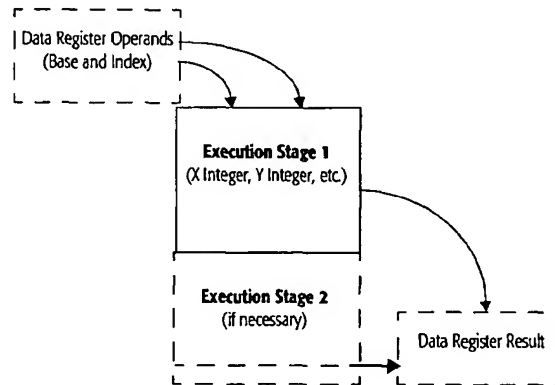
The dedicated integer execution unit contained within the register Y execution pipeline can execute the basic word and doubleword ALU operations (ADD, AND, CMP, OR, SUB and XOR), zero-extend, and sign-extend operations. Data register results are available after one clock.

The register X and Y execution pipelines each contain a dedicated MMX execution unit that handles add/subtract, logical, and pack/unpack MMX instructions. The multimedia ALU units are symmetrical and can be used simultaneously. This means that the processor can execute 2 multimedia ALU cycles each clock cycle.

A number of execution resources are available to both the register X and Y execution pipelines. These shared resources include the MMX shifter, 3D ALU, and the combined MMX/3D multiplier. Figure 50 on page 91 shows which instruction types are associated with the various execution pipelines.

Any combination of two operations that do not utilize the same shared execution resource can be issued and executed simultaneously. For example, the following pairs of register operations can execute together—MMX logical and 3D add, 3D add and 3D multiply, MMX multiply and 3D add, etc. If issued simultaneously, the following examples result in resource contentions and the stall of one RISC86 operation: MMX multiply and 3D multiply, two MMX multiplies, two 3D multiplies, two 3D adds, etc.

Figure 116 shows the data flow architecture of the single-stage or double-stage integer execution unit pipeline. There are few operations (such as integer multiply) that require a second execution stage. The operation issue and operand fetch stage (execution stage 0) that precede this execution stage are not part of the execution pipeline. The data register result is produced near the end of the execution pipe stage.



**Figure 116. Register X and Y Execution Stages**

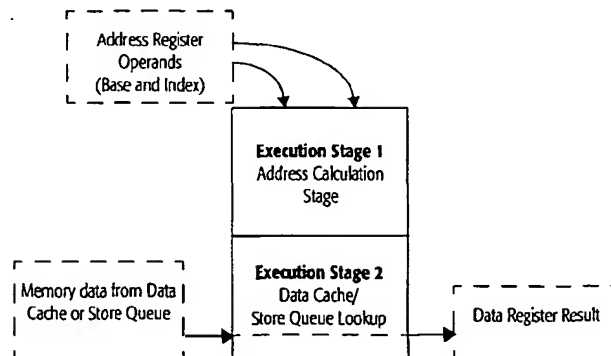
## **B** Code Optimization

### **Load Unit**

The load unit is a two-stage pipelined design that performs data memory reads. It has a two-clock latency from the time it receives the address register operands until it produces a data register result on a data cache hit. A cache miss produces longer latencies. The load unit and the data cache support hit-under-miss operations where a load operation bypasses a previous load operation that is stalled waiting for a cache line refill. This unit uses two address register operands and a memory data value as inputs, and produces a data register result.

Memory read data can come from either the data cache or from the store queue entry (for a recent store). If the data is forwarded from the store queue, there is zero additional execution latency, which means that a dependent load operation can complete its execution one clock after a store operation completes execution.

Figure 117 shows the architecture of the two-stage load execution pipeline. The address register operands are received at the end of the operand fetch pipe stage, and the data register result is produced near the end of the second execution pipe stage. The operation issue and fetch stages that precede this execution stage are not shown.



**Figure 117. Load Execution Unit**

## Store Unit

The store execution unit is a two-stage pipelined design that performs data memory writes, and, in some cases, produces an address register result. For inputs, the store unit uses two address register operands and, during actual memory writes, a store data register operand. This unit also produces an address register result for some store unit operations. For most store operations, for example those that write data to memory, the store unit produces a physical memory address and the associated data bytes to be written. After execution completes, these results are entered in a new store queue entry. The store queue can hold up to seven data results, each of which can be 64 bits.

The store unit has a one-clock execution latency from the time it receives address register operands until the time it produces an address register result. The most common examples are the Load Effective Address (Lea) and Store and Update (Push) RISC86 operations, which are produced from the x86 LEA and PUSH instructions, respectively. Most store operations do not produce an address register result and only perform a memory write. The Push operation is unique because it produces an address register result and performs a memory write.

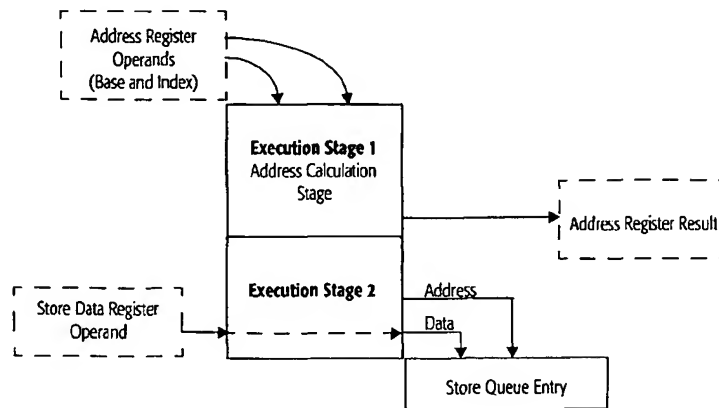
The store unit has a one-clock execution latency from the time it receives address register operands until it enters the store memory address and data pair into the store queue.

The store unit can have a three-clock latency from the time it receives address register operands and a store data register operand until it enters the memory address and data pair into the store queue.

*Note: Address register operands are required at the start of execution, but register store data is not required until the end of execution.*

Figure 118 on page 464 shows the architecture of the two-stage store execution pipeline. The operation issue and fetch stages that precede this execution stage are not part of the execution pipeline. The address register operands are received at the end of the operand fetch pipe stage, and the new store queue entry is created upon completion of the second execution pipe stage.

## **B** Code Optimization



**Figure 118. Store Unit Execution Pipeline**

### **Branch Condition Unit**

The branch condition unit is separate from the branch prediction logic, which is utilized at x86 instruction decode time. This unit resolves conditional branches, such as JCC and LOOP instructions, at a rate of up to one per clock cycle. This unit has a dedicated RISC86 issue bus from the scheduler. For more information, see “Branch-Prediction Logic” on page 21.

### **Floating-Point Unit**

The floating-point unit (FPU) handles all register operations for x87 instructions. The execution unit is a single-stage design that takes data register operands as inputs and produces a data register result as an output. The most common floating-point instructions have a two clock execution latency from the time the FPU receives data register operands until it produces a data register result. The FPU has its own RISC86 issue bus from the scheduler. For more information, see Chapter 9, “Floating-Point and Multimedia Execution Units” on page 253.

## Latencies and Throughput

Table 86 summarizes the static latencies and throughput of each execution unit. Knowing instruction latencies is important when figuring critical instruction dependencies.

**Table 86. RISC86 Execution Latencies and Throughput**

Execution Unit	Operations	Latency	Throughput
Register X Integer Unit	Integer ALU	1	1
	Integer Multiply	2–3	2–3
	Integer Shift	1	1
Register X Multimedia Unit	MMX Add/Subtract	1	1
	MMX Logical, Pack, Unpack	1	1
Register Y Integer Unit	Integer ALU (16- and 32- bit operands)	1	1
Register Y Multimedia Unit	MMX Add/Subtract	1	1
	MMX Logical, Pack, Unpack	1	1
Multimedia/3D Shared Execution Units (X and Y)	MMX Shifter	1	1
	MMX/3D Multiply, Reciprocal and, Reciprocal Square Root Iteration	2	1
	3D Add, Compare, Integer Conversion, Reciprocal, and Reciprocal Square Root Table Lookup	2	1
Load	From Address Register Operands to Data Register Result	2	1
	Memory Read Data from Data Cache/Store Queue to Data Register Result	0	1
Store	From Address Register Operands to Address Register Result	1	1
	From Store Data Register Operands to Store Queue Entry	1	1
	From Address Register Operands to Store Queue Entry	3	1
Branch	Resolves Branch Conditions	1	1
FPU	FADD, FSUB	2	2
	FMUL	2	2
<b>Note:</b> No additional latency exists between execution of dependent operations. Bypassing of register results directly from producing execution units to the operand inputs of dependent units is fully supported. Similarly, forwarding of memory store values from the store queue to dependent load operations is supported.			

# **B** Code Optimization

## **Resource Constraints**

To optimize code effectively, execution resource constraints must be considered. Due to a fixed number of execution units, even with up to six RISC86 operations per cycle, optimal execution parallelism should be carefully scheduled.

For example, if an IMUL is decoded and issued to the X pipeline, for the next two to three cycles integer, MMX, and 3D RISC86 operations can only be issued to the Y pipeline. Another example is two ALU instructions that require the load unit. Only one load can occur each cycle, therefore, one instruction would stall for a cycle.

Contention for execution resources can cause delays in the issuing and execution of instructions. In addition, stalls due to resource constraints can increase dependency latencies to cause or exacerbate stalls due to dependencies. In general, constraints that delay non-critical instructions do not impact performance because such stalls typically overlap with the execution of critical operations.

## **Code Sample Analysis**

The samples in this section show the execution behavior of several series of instructions as a function of decode constraints, dependencies, and execution resource constraints.

*Note: These samples are animated in the AMD-K6 3D simulator available on the CD-ROM that accompanies this book.*

The sample tables show the x86 instructions, the RISC86 operation equivalents, the clock counts, and a description of the events occurring within the processor.

The following nomenclature is used to describe the current location of a RISC86 operation (RISC86op):

- D — Decode stage
- I<sub>X</sub> — Issue stage of register X unit
- O<sub>X</sub> — Operand fetch stage of register X unit
- E<sub>X1</sub> — Execution stage 1 of register X unit
- E<sub>X2</sub> — Execution stage 2 of register X unit

- $I_Y$  — Issue stage of register Y unit
- $O_Y$  — Operand fetch stage of register Y unit
- $E_{Y1}$  — Execution stage 1 of register Y unit
- $E_{Y2}$  — Execution stage 2 of register Y unit
- $I_L$  — Issue stage of load unit
- $O_L$  — Operand fetch stage of load unit
- $E_{L1}$  — Execution stage 1 of load unit
- $E_{L2}$  — Execution stage 2 of load unit
- $I_S$  — Issue stage of store unit
- $O_S$  — Operand fetch stage of store unit
- $E_{S1}$  — Execution stage 1 of store unit
- $E_{S2}$  — Execution stage 2 of store unit

*Note: Instructions execute more efficiently (that is, without delays) when scheduled apart by suitable distances based on dependencies. In general, the samples in this section show poorly scheduled code in order to illustrate the resultant effects.*



# B Code Optimization

**Table 87. Sample 1 – Integer Register Operations**

Instruction Number	Instruction	RISC86 Opcodes	Clocks								
			1	2	3	4	5	6	7	8	9
1	IMUL EAX, EBX	alux	D	D	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>				
		alux				I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>			
		alux					I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>		
2	INC ESI	alu			D	I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>			
3	MOV EDI, 0x07F4	limm			D						
4	SHL EAX, 8	alux				D		I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>	
5	OR EAX, 0x0F	alu				D	I <sub>Y</sub>	O <sub>Y</sub>	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>
6	ADD ESI, EDX	alu					D	I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>	
7	SUB EDI, ECX	alu					D		I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>
<b>Comments for Each Instruction Number</b>											
1	It takes two decode cycles because IMUL is vector decoded. The IMUL instruction is executable only in the integer X unit. It is a non-pipelined 2–3 cycle latency register operation that is equivalent to three serially-dependent register operations (the result of the second and third operations are AX and DX, respectively).										
2	This simple alu operation ends up in the Y pipe.										
3	A load immediate (limm) RISC86 operation does not require execution. The result value is immediately available to dependent operations.										
4	Shift instructions are only executable in the integer X unit. Issue is delayed by preceding IMUL operations due to a resource constraint of the integer X unit.										
5	The register operation is bumped out of the integer Y unit in clock 6 because it must wait for more than one cycle for its dependencies to resolve. It is reissued in the next cycle to the integer X unit (just in time for availability of its operands).										
6	This add alu falls through to the integer Y unit right behind the first issuance of instruction #5 without delay (as a result of instruction #5 being bumped out of the way).										
7	The issuance of the subtract register operation is delayed in clock 6 due to the resource constraints of the integer Y unit.										

Table 88. Sample 2 – Integer Register and Memory Load Operations

Instruction Number	Instruction	RISC86 Opcodes	Clocks										
			1	2	3	4	5	6	7	8	9	10	11
1	DEC EDX	alu	D	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>							
2	MOV EDI, [ECX]	load	D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>						
3	SUB EAX, [EDX+20]	load		D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>					
		alu			I <sub>X</sub>	O <sub>X</sub>	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>				
4	SAR EAX, 5	alux		D		I <sub>X</sub>	O <sub>X</sub>	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>			
5	ADD ECX, [EDI+4]	load			D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>				
		alu				I <sub>Y</sub>	O <sub>Y</sub>	I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>			
6	AND EBX, 0x1F	alu			D		I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>				
7	MOV ESI, [0x0F100]	load				D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>			
8	OR ECX, [ESI+EAX*4+8]	load				D		I <sub>L</sub>	O <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>	
		alu							I <sub>X</sub>	O <sub>X</sub>	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>
Comments for Each Instruction Number													
1	This simple alu operation ends up in the X pipe.												
2	This operation occupies the load execution unit.												
3	The register operand for the load operation is bypassed, without delay, from the result of instruction #1's register operand. In clock 4, the register operation is bumped out of the integer X unit while waiting for the previous load operation result to complete. It is reissued just in time to receive the bypassed result of the load.												
4	Shift instructions are only executable in the integer X unit. The register operation is bumped in clock 5 while waiting for the result of the preceding instruction #3.												
5	The register operand for the load operation is bypassed, without delay, from the result of instruction #2's register operand. This and most surrounding load operations are generated by instruction decoders, and issued and smoothly executed by the load unit at a rate of one clock per cycle. In clock 5, the register operation is bumped out of the integer Y unit while waiting for the previous load operation result to complete.												
6	The register operation falls through into the integer Y unit right behind instruction #5's register operation.												
7	This operation falls into the load unit behind the load in instruction #5.												
8	The operand fetch for the load operation is delayed because it needs the result of the immediately preceding load operation #7 as well as the results from earlier instructions #3 and #4.												

# B Code Optimization

**Table 89. Sample 3 – Integer Register and Memory Load/Store Operations**

Instruction Number	Instruction	RISC86 Opcodes	Clocks										
			1	2	3	4	5	6	7	8	9	10	11
1	MOV EDX, [0xA0008F00]	load	D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>						
2	ADD [EDX+16], 7	load		D	I <sub>L</sub>	O <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>				
		alu			I <sub>X</sub>	O <sub>X</sub>	I <sub>X</sub>	O <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>			
		store			I <sub>S</sub>	O <sub>S</sub>	O <sub>S</sub>	E <sub>S1</sub>	E <sub>S2</sub>	E <sub>S2</sub>			
3	SUB EAX, [EDX+16]	load			D	I <sub>L</sub>	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>	E <sub>L2</sub>		
		alu				I <sub>X</sub>	O <sub>X</sub>	I <sub>X</sub>	I <sub>X</sub>	O <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>	
4	PUSH EAX	store			D	I <sub>S</sub>	I <sub>S</sub>	O <sub>S</sub>	E <sub>S1</sub>	E <sub>S2</sub>	E <sub>S2</sub>	E <sub>S2</sub>	
5	LEA EBX, [ECX+EAX*4+3]	store				D		I <sub>S</sub>	O <sub>S</sub>	O <sub>S</sub>	O <sub>S</sub>	E <sub>S1</sub>	E <sub>S2</sub>
6	MOV EDI, EBX	alu				D	I <sub>Y</sub>	O <sub>Y</sub>	I <sub>Y</sub>	O <sub>Y</sub>	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>
<b>Comments for Each Instruction Number</b>													
1	This operation occupies the load unit.												
2	This long-decoded ADD instruction takes a single clock to decode. The operand fetch for the load operation is delayed waiting for the result of the previous load operation from instruction #1. The store operation completes concurrent with the register operation. The result of the register operation is bypassed directly into a new store queue entry created by the store operation.												
3	The issue of the load operation is delayed because the operand fetch of the preceding load operation from instruction #2 was delayed. The completion of the load operation is held up due to a memory dependency on the preceding store operation of instruction #2. The load operation completes immediately after the store operation, with the store data being forwarded from a new store queue entry.												
4	Completion of the store operation is held up due to a data dependency on the preceding instruction #3. The store data is bypassed directly into a new store queue entry from the result of instruction #3's register operation.												
5	The Lea RISC86 operation is executed by the store unit. The operand fetch is delayed waiting for the result of instruction #3. The register result value is produced in the first execution stage of the store unit.												
6	This simple alu operation is stalled due to the dependency of the BX result in instruction #5.												

Table 90. Sample 4 – Integer, MMX, and Memory Load/Store Operations

Inst. Num.	Instruction	RISC86 Opcodes	Clocks											
			1	2	3	4	5	6	7	8	9	10	11	12
1	PADDSW MM0, MM4	alu	D	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>								
2	PADDSW MM1, MM5	alu	D	I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>								
3	PSRAW MM0, 3	alu		D	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>							
4	MOVQ MM2, [EAX+EBX]	mload		D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>						
5	PAND MM0, MM3	alu			D	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>						
6	PMULLW MM2, [EDI+8]	mload			D	I <sub>L</sub>	O <sub>L</sub>	E <sub>L1</sub>	E <sub>L2</sub>					
		alu				I <sub>Y</sub>	O <sub>Y</sub>	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>	E <sub>X2</sub>			
7	MOVQ [ESP+4], MM2	mstore				D	I <sub>S</sub>	O <sub>S</sub>	E <sub>S1</sub>	E <sub>S2</sub>	E <sub>S2</sub>			
8	ADD EBX, ECX	alu				D	I <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>					
9	PMULLW MM6, MM7	alu					D	I <sub>Y</sub>	O <sub>Y</sub>	E <sub>Y1</sub>	E <sub>Y1</sub>	E <sub>Y2</sub>		
10	PMADDWD MM2, MM6	alu					D		I <sub>X</sub>	O <sub>X</sub>	O <sub>X</sub>	O <sub>X</sub>	E <sub>X1</sub>	E <sub>X2</sub>

**Comments for Each Instruction Number**

- 1, 2 Instructions 1 and 2 are decoded, issued, and executed simultaneously and in parallel due to no decode restrictions, dependency delays, or execution resource constraints.
- 3 This instruction is decoded, issued, and executed without delay, one cycle behind the preceding one-cycle execution latency instruction on which it is dependent.
- 4 This multimedia operation occupies the load unit.
- 5 This instruction is decoded, issued, and executed without delay, right behind the preceding operations on which it is dependent.
- 6 This and the preceding instruction are decoded and issued together without delay. The operand fetch of the register operation is delayed because of the dependency on the associated load. As a result, the register operation is bumped out of register unit Y in clock 5 and is reissued in the next cycle to register unit X (as it happens), just in time for availability of its operands.
- 7 Completion of this store operation is held up due to a data dependency on the preceding MMX multiply register operation (which has a two-cycle execution latency). The store data is bypassed directly into a new store queue entry from the result of the register operation.
- 8 This operation is issued to register unit X and executes without delay and out-of-order with respect to the preceding register operation from instruction #6 (which was bumped out of the way while waiting for its operands).
- 9 This MMX multiply register operation issues to and starts execution in register unit Y in parallel with an MMX multiply register operation from instruction #6 which simultaneously issues to and starts execution in register unit X. Due to an execution resource constraint, this operation is delayed one cycle in its first execution pipe stage and then executes and completes normally, one cycle behind the other contending register operation. (This takes advantage of the pipelined nature of the MMX multiply execution logic.)
- 10 The issue of this operation is delayed (in clock 6) for one cycle due to two earlier register operations being selected for issue. It is then delayed further during operand fetch while waiting for the preceding two-cycle latency MMX multiply register operations to complete execution.

# **B** *Code Optimization*

---

## **Optimization Coding Guidelines**

---

### **General x86 Optimization Techniques**

This section describes general code optimization techniques specific to superscalar processors (that is, techniques common to the AMD-K6 3D processor, AMD-K5™ processor, and Pentium-family processors). In general, all optimization techniques used for the AMD-K5 processor, Pentium, and Pentium Pro processors either improve the performance of the AMD-K6 3D processor or are not required and have a neutral effect (usually due to fewer coding restrictions with the AMD-K6 3D processor).

**Short Forms**—Use shorter forms of instructions to increase the effective number of instructions that can be examined for decoding at any one time. Use 8-bit displacements and jump offsets where possible.

**Simple Instructions**—Use simple instructions with hardwired decode (pairable, short, or fast) because they perform more efficiently. This includes “register←register op memory” as well as “register←register op register” forms of instructions.

**Dependencies**—Spread out true dependencies to increase the opportunities for parallel execution. Anti-dependencies and output dependencies do not impact performance.

**Memory Operands**—Instructions that operate on data in memory (load/operation/store) can inhibit parallelism. The use of separate move and ALU instructions allows better code scheduling for independent operations. However, if there are no opportunities for parallel execution, use the load/operation/store forms to reduce the number of register spills (storing values in memory to free registers for other uses).

**Register Operands**—Maintain frequently used values in registers rather than in memory.

**Stack References**—Use ESP for stack references so that EBP remains available.

**Stack Allocation**—When allocating space for local variables and/or outgoing parameters within a procedure, adjust the stack pointer and use moves rather than pushes. This method of allocation allows random access to the outgoing parameters so that they can be set up when they are calculated instead of being held somewhere else until the procedure call. This method also reduces ESP dependencies and uses fewer execution resources.

**Data Embedding**—When data is embedded in the code segment, align it in separate cache blocks from nearby code. This technique avoids some overhead when maintaining coherency between the instruction and data caches.

**Loops**—Unroll loops to get more parallelism and reduce loop overhead, even with branch prediction. Inline small routines to avoid procedure-call overhead. For both techniques, however, consider the cost of possible increased register usage, which might add load/store instructions for register spilling. Unrolling large code loops can result in the inefficient use of L1 instruction caches.

**Code Alignment**—Aligning subroutines at 0-mod-16 (or ideally, at 0-mod-32) address boundaries optimizes instruction cache-fill efficiency. Keeping the starting point of loops at least two instructions away from the end of 32-byte cache lines optimizes branch-target instruction fetch and decode efficiency.

## **B** Code Optimization

### **General AMD-K6 3D Processor x86 Coding Optimizations**

This section describes general code optimization techniques specific to the AMD-K6 processor models 6, 7, 8, and 9.

**Use short-decodeable instructions**—To increase decode bandwidth and minimize the number of RISC86 operations per x86 instruction, use short-decodeable x86 instructions. See instructions labeled as 'short' in the 'Decode Type' column in Tables 12 through 15 starting on page 53.

**Pair short-decodeable instructions**—Two short-decodeable x86 instructions can be decoded per clock, using the full decode bandwidth of the processor.

*Note: For the AMD-K6 3D processor, all MMX and 3D instructions are short-decodeable except the EMMS, FEMMS, and PREFETCH instructions.*

**Avoid using complex instructions**—The more complex and uncommon instructions are vector decoded and can generate a larger ratio of RISC86 operations per x86 instruction compared with short-decodeable or long-decodeable instructions.

**Avoid multiple and accumulated prefixes**—In order to accomplish an instruction decode, the decoders require sufficient predecode information. When an instruction has multiple prefixes and this cannot be deduced by the decoders (due to a lack of data in the instruction decode buffer), the first decoder retires and accumulates one prefix per cycle until the instruction is completely decoded. Table 91 on page 475 shows when prefixes are accumulated and decoding is serialized.

**Table 91. Decode Accumulation and Serialization**

Decode #1	Decoder #2	Results
Instruction		Single instruction decoded.
Instruction	Instruction	Dual instruction decode.
Instruction	Prefix	Single instruction decode and prefix is accumulated.
Prefix	Instruction (modified by Prefix)	No prefix accumulation and single instruction is decoded.
PrefixA	PrefixB	Accumulate PrefixA and cancel decode of the second prefix.
PrefixB	Instruction	If a prefix has already been accumulated in the previous decode cycle, accumulate PrefixB and cancel instruction decode, wait for next decode cycle to decode the instruction.

**0Fh prefix usage**—0Fh does not count as a prefix for the decoder accumulation rules (that is, it does not cause accumulation).

**Avoid long instruction length**—Use x86 instructions that are less than eight bytes in length. An x86 instruction that is longer than seven bytes cannot be short-decoded.

**Use read-modify-write instructions over discrete equivalent**—No advantage is gained by splitting read-modify-write instructions into a load-execute-store instruction group. Both read-modify-write instructions and load-execute-store instruction groups decode and execute in one cycle but read-modify-write instructions promote better code density.

**Move rarely used code and data to separate pages**—Placing code, such as exception handlers, in separate pages and data, such as error text messages, in separate pages maximizes the use of the TLBs and prevents table pollution with rarely used items.

**Avoid mixing code size types**—Size prefixes that affect the length of an instruction can sometimes inhibit dual decoding.

**Always pair CALL and RETURN**—If CALLs and RETs are not paired, the return address stack gets out of synchronization, increasing the latency of returns and decreasing performance.



## **B** Code Optimization

**Exploit parallel execution of integer and floating-point multiplies**—The AMD-K6 3D processor allows simultaneous integer and floating-point multiplies using separate, low-latency multipliers.

**Avoid more than 16 levels of nesting in subroutines**—More than 16 levels of nested subroutine calls overflow the return address stack, leading to lower performance. While this is not a problem for most code, recursive subroutines might easily exceed 16 levels of subroutine calls. If the recursive subroutine is tail recursive, it can usually be mechanically transformed into an iterative version, which leads to increased performance.

**Place frequently used stack data within 128 bytes of the EBP**—The statically most-referenced data items in a function's stack frame should be located from -128 to +127 bytes from EBP. This technique improves code density by enabling the use of an 8-bit sign-extended displacement instead of a 32-bit displacement.

**Avoid superset dependencies**—Using the larger form of a register immediate after an instruction uses the smaller form creates a superset dependency and prevents parallel execution. For example, avoid the following type of code:

```
OR    AH,07h
ADD   EAX,1555555h
```

One method for avoiding superset dependencies is to schedule the instruction with the superset dependency (for example, the ADD instruction) 4–6 instructions later than would normally be preferable. Another method, useful in some cases, is to use the MOVZX instruction to efficiently convert a byte-size value to a doubleword-size value, which can then be combined with other values in 32-bit operations.

**Avoid excessive loop unrolling or code inlining**—Excessive loop unrolling or code inlining increases code size and reduces locality, which leads to lower cache hit rates and reduced performance.

**Avoid splitting a 16-bit memory access in 32-bit code**—No advantage is gained by splitting a 16-bit memory access in 32-bit code into two byte-sized accesses. This technique avoids the operand size override.

**Avoid data dependent branches around a single instruction—** Data dependent branches acting upon basically random data cause the branch prediction logic to mispredict the branch about 50% of the time. Design branch-free alternative code sequences. The effect is shorter average execution time. The following examples illustrate this concept:

■ **Signed integer ABS function ( $x = \text{labs}(x)$ )**

Static Latency: 4 cycles

```
MOVCX, [x] ;load value
MOVEBX, ECX
SARCCX, 31
XOREBX, ECX ;1's complement if x<0, else don't modify
SUBEBX, ECX ;2's complement if x<0, else don't modify
MOV[x], EBX ;save labs result
```

**Unsigned integer min function ( $z = x < y ? x : y$ )**

Static Latency: 4 cycles

```
MOVEAX, [x] ;load x value
MOVEBX, [y] ;load y value
SUBEBX, EBX ;set carry flag if y is greater than x
SBBECX, ECX ;get borrow out from previous SUB
ANDECX, EAX ;if x > y, ECX = x-y, else 0
ADDECX, EBX ;if x > y, return x-y-y = x, else y
MOV[z], ECX ;save min (x,y)
```

■ **Hexadecimal to ASCII conversion**

( $y = x < 10 ? x + 0x30 : x + 0x41$ )

Static Latency: 4 cycles

```
MOVAL, [x] ;load x value
CMPAL, 10 ;if x is less than 10, set carry flag
SBBAL, 69h ;0..9 -> 96h, Ah..Fh -> A1h...A6h
DAS ;0..9: subtract 66h, Ah..Fh: Subtract 60h
MOV[y], AL ;save conversion in y
```

**Avoid using the [ESI] addressing mode—** This addressing mode forces the instructions using it to become vector decoded. There are two ways to avoid this problem. The first way is to use another register. The second way is to alter the addressing mode by explicitly coding [ESI+0]. Assemblers may optimize this to [ESI] by removing the 0.

# **B** *Code Optimization*

## **AMD-K6 3D Processor Integer x86 Coding Optimizations**

This section describes integer code optimization techniques specific to the AMD-K6 processor models 6, 7, 8, and 9.

**Neutral code filler**—Use the XCHG EAX, EAX or NOP instruction when aligning instructions. XCHG EAX, EAX consumes one decode slot but requires no execution resources. Essentially, the scheduler absorbs the equivalent RISC86 operation without requiring any of the execution units.

**Inline REP String with low counts**—Expand REP String instructions into equivalent sequences of simple x86 instructions. This technique eliminates the setup overhead of these instructions and increases instruction throughput.

**Use ADD reg, reg instead of SHL reg, 1**—This optimization technique allows the scheduler to use either of the two integer adders rather than the single shifter and effectively increases overall throughput. The only difference between these two instructions is the setting of the AF flag.

**Use MOVZX and MOVSX to zero-extend and sign-extend byte-size and word-size operands to doubleword length**—For example, typical code for zero extension creates a superset dependency when the zero-extended value is used, as in the following code:

```
XOR    EAX,EAX
MOV    AL,[mem]
```

Instead, use the following code:

```
MOVZX  EAX,BYTE PTR [mem]
```

**Use load-execute integer instructions**—Most load-execute integer instructions are short-decodeable and can be decoded at the rate of two per cycle. Splitting a load-execute instruction into two separate instructions—a load instruction and a reg, reg instruction—reduces decoding bandwidth and increases register pressure. The split-instruction form can be used to avoid scheduler stalls for longer executing instructions and to explicitly schedule the load and execute operations.

**Use AL, AX, and EAX to improve code density**—In many cases, instructions using AL and EAX can be encoded in one less byte than using a general-purpose register. For example, ADD AX, 0x5555 should be encoded 05 55 55 and not 81 C0 55 55.

**Clear registers using MOV reg, 0 instead of XOR reg, reg**—Executing XOR reg, reg requires additional overhead due to register dependency checking and flag generation. Using MOV reg, 0 produces a limm (load immediate) RISC86 operation that is completed when placed in the scheduler and does not consume execution resources.

**Use 8-bit sign-extended immediates**—Using 8-bit sign-extended immediates improves code density with no negative effects on the processor. For example, ADD BX, -55 should be encoded 83 C3 FB and not 81 C3 FF FB.

**Use 8-bit sign-extended displacements for conditional branches**—Using short, 8-bit sign-extended displacements for conditional branches improves code density with no negative effects on the processor.

**Use integer multiply over shift-add sequences when it is advantageous**—The AMD-K6 3D processor features a low-latency integer multiplier. Therefore, almost any shift-add sequences can have higher latency than MUL or IMUL instructions. An exception is a trivial case involving multiplication by powers of two by means of left shifts. In general, replacements should be made if the shift-add sequences have a latency greater than or equal to 3 clocks.

**Carefully choose the best method for pushing memory data**—To reduce register pressure and code dependency, use PUSH [mem] rather than MOV EAX, [mem], PUSH EAX.

**Balance the use of CWD, CBW, CDQ, and CWDE**—These instructions require special attention to avoid either decreased decode or execution bandwidth. The following code illustrates the possible trade-offs:

- The following code replacement trades decode bandwidth (CWD is vector decoded, but with only one RISC86 operation) with execution bandwidth (SAR requires two RISC86 operations, including a shift):

```
Replace:CWD With: MOV    DX,AX
                  SAR     DX,15
```

## **B** Code Optimization

- The following code replacement improves decode bandwidth (CBW is vector decoded while MOVSB is short decoded):

Replace:CBW With: MOVSB AX,AL

- The following code replacement trades decode bandwidth (CDQ is vector decoded, but with only two RISC86 operations) with execution bandwidth (SAR requires two RISC86 operations, including a shifter):

Replace:CDQ With: MOV EDX, EAX  
SAR EDX, 31

- The following code replacement improves decode bandwidth (CWD is vector decoded while MOVSB is short decoded):

Replace:CWD With: MOVSB EAX, AX

**Replace integer division by constants with multiplication by the reciprocal**—This optimization is commonly used on RISC processors. Because the AMD-K6 3D processor has an extremely fast integer multiply (two cycles) and the integer division delivers only two bits of quotient per cycle (approximately 18 cycles for 32-bit divides), the equivalent code is much faster. The following examples illustrate the use of integer division by constants:

- **Unsigned division by 10 using multiplication by reciprocal**  
Static Latency: 5 cycles

```
; IN:EAX = dividend
; OUT:EDX = quotient
MOVEDX, 0CCCCCDh; 0.1 * 2^32 * 8 rounded up
MULEDX
SHRDX, 3           ; divide by 2^32 * 8
```

- **Unsigned division by 3 using multiplication by reciprocal**  
Static Latency: 5 cycles

```
; IN:EAX = dividend
; OUT:EDX = quotient
MOVEDX, 0AAAAAAh; 1/3 * 2^32 * 2 rounded up
MULEDX
SHRDX, 1           ; divide by 2^32 * 2
```

- **Signed division by 2**  
Static Latency: 3 cycles

```
; IN:EAX = dividend
; OUT:EAX = quotient
CMPEAX, 80000000h; CY = 1, if dividend >= 0
SBBEAX, -1       ; increment dividend if it is < 0
SAREAX, 1        ; perform a right shift
```

- **Signed division by  $2^n$**   
**Static Latency: 5 cycles**  

```

; IN:EAX = dividend
; OUT:EAX = quotient
MOVEDX, EAX      ;sign extend into EDX
SARL, 31          ;EDX = 0xFFFFFFFF if dividend < 0
ANDL, (2^n-1)     ;mask correction (use divisor -1)
ADDEAX, EDX       ;apply correction if necessary
SAR EAX, (n)       ;perform right shift by log2 (divisor)

```
- **Signed division by -2**  
**Static Latency: 4 cycles**  

```

; IN:EAX = dividend
; OUT:EAX = quotient
CMPEAX, 80000000h;CY = 1, if dividend >=0
SBBEAX, -1        ;increment dividend if it is <0
SARL, 1           ;perform right shift
NEG EAX           ;use (x/-2) = -(x/2)

```
- **Signed division by  $-(2^n)$**   
**Static Latency: 6 cycles**  

```

; IN:EAX = dividend
; OUT:EAX = quotient
MOVEDX, EAX      ;sign extend into EDX
SARL, 31          ;EDX = 0xFFFFFFFF if dividend < 0
ANDL, (2^n-1)     ;mask correction (-divisor -1)
ADDEAX, EDX       ;apply correction if necessary
SARL, (n)         ;right shift by log2(-divisor)
NEG EAX           ;use (x/-(2^n)) = -(x/2^n)

```
- **Remainder of signed integer 2 or (-2)**  
**Static Latency: 4 cycles**  

```

; IN:EAX = dividend
; OUT:EDX = quotient
MOVEDX, EAX      ;sign extend into EDX
SARL, 31          ;EDX = 0xFFFFFFFF if dividend < 0
ANDL, 1           ;compute remainder
XOREAX, EDX       ;negate remainder if
SUB EAX, EDX      ;dividend was < 0
MOV[quotient], EAX

```
- **Remainder of signed integer ( $2^n$ ) or  $-(2^n)$**   
**Static Latency: 6 cycles**  

```

; IN:EAX = dividend
; OUT:EDX = quotient
MOVEDX, EAX      ;sign extend into EDX
SARL, 31          ;EDX = 0xFFFFFFFF if dividend < 0
ANDL, (2^n-1)     ;mask correction (abs(divisor)-1)
ADDEAX, EDX       ;apply pre-correction
ANDL, (2^n-1)     ;mask out remainder (abs(divisor)-1)
SUB EAX, EDX      ;apply pre-correction if necessary
MOV[quotient], EAX

```

---

# **B** Code Optimization

---

## **AMD-K6 3D Processor Multimedia Coding Optimizations**

This section describes multimedia code optimization techniques for the AMD-K6 3D processor.

**For optimal floating-point performance**—Wherever possible, use the packed single-precision, floating-point capability of 3D technology instead of the single-precision, double-precision, and extended-precision floating-point capabilities of the x87 floating-point unit. The 3D units are fully pipelined, allow vectorized optimizations, are not stack based, and provide faster inverse, square root, and inverse square root calculations.

**Issues to ensure optimal predecode of MMX and 3D instructions**—Attention must be paid to coding issues that can inhibit the predecode, and later dual decode, of x86 instructions. Instructions are predecoded during instruction cache line fills. The predecode information that is produced and then stored in the predecode cache is later used by the instruction decoders to quickly find consecutive instructions and, therefore, enable dual-instruction decode. (The predecode information, in particular, reflects the length of instructions.)

The processor predecode scheme is based on a number of assumptions and constraints that have been mentioned previously, but which are repeated here for convenience:

- Only a subset of x86 instructions are short decodeable and require predecode information. These include all MMX and 3D instructions except for the EMMS, FEMMS, and PREFETCH instructions.
- Predecodeable instructions can be up to seven bytes in length.
- The processor predecoders can only examine the first three bytes of an instruction to determine the length of the instruction and generate the predecode information. To determine instruction length, non-modR/M instructions require examination of the opcode byte, and modR/M instructions require the examination of the opcode byte plus the modR/M byte. Instructions with a 0Fh prefix require the examination of the 0Fh byte in addition to the opcode byte and any modR/M byte. Finally, modR/M address modes with a sib byte and no displacement (modR/M = 00\_xxx\_100b) require examination of the

additional sib byte. Instructions in this last category that also require a 0Fh prefix violate the three-byte predecode constraint and, therefore, cannot be predecoded—these instructions use either [disp32 + (scaled)index] or [base + (scaled)index] address modes and, therefore, require the examination of four bytes to determine instruction length.

- The 32-bit modR/M address mode [ESI] cannot be predecoded.
- For instructions starting within the last two bytes of a cache line, the predecode logic is not able to scan past the end of the cache line when it needs to examine more bytes to determine the length of an instruction. This constraint limits the type of instructions that can be predecoded at the end of a cache line. For example, a modR/M instruction that starts on the last byte of a 32-byte cache line, or a 0Fh-prefix plus modR/M instruction that starts within the last two bytes of the cache line, cannot be predecoded.
- MMX and 3D instructions have a 0Fh-prefix byte, an opcode byte, and a modR/M byte, all of which must be examined by the predecode logic.

These constraints result in the following recommendations for successful predecode of multimedia instructions:

- With 3D instructions, do not use address modes with large (32-bit) displacements. Large displacements result in a total instruction length of eight bytes (including the additional suffix byte used at the end of the instruction as a sub-opcode byte).
- With MMX and 3D instructions, do not use the [disp32 + (scaled)index], [base + (scaled)index], or [ESI] address modes.
- Avoid placing the start of MMX and 3D instructions in the last two bytes of a cache line. If not successfully predecoded, MMX instructions default to vector decodes and 3D instructions default to long decodes.

A comparison of the instruction decode clock-cycle count on optimized code is as follows:

- 0.5 cycle for one short decode as part of a dual decode.
- 1.0 cycle for a single long decode.
- 2.0 cycles for a single vector decode (for simple instructions such as MMX and 3D instructions).



## **B** *Code Optimization*

**Avoid using MMX/3D registers to move double-precision floating-point data**—Although using an MMX/3D register to move x87 floating-point data appears fast, using these registers requires the use of the EMMS or FEMMS instruction when switching from MMX or 3D instructions to x87 instructions.

**Use the FEMMS instruction instead of the EMMS instruction**—The AMD-K6 3D processor implements an improved version of the EMMS instruction, called FEMMS. Because the MMX/3D registers are mapped onto the x87 stack, an EMMS or FEMMS instruction must be executed when switching from MMX or 3D code to x87 code. Execution of the EMMS or FEMMS instruction marks the floating-point tag word as empty (all 1's), which guarantees correct x87 results and ensures that no x87 exceptions occur in the subsequent code due to a stack overflow.

Each time the processor encounters a switch between MMX or 3D code and x87 code, in either direction, a significant clock-cycle count penalty occurs. The FEMMS instruction was created to reduce this penalty. The FEMMS instruction sets the floating-point tag word to empty (like EMMS), and also sets all of the register values as undefined. If a switch is required following a FEMMS instruction, it executes in less than half the cycles required after an EMMS instruction. The switch overhead occurs when an x87 instruction is encountered, and not during the execution of the EMMS and FEMMS instructions. In addition, the FEMMS instruction executes in 3 clock cycles, 2 cycles less than the EMMS instruction. For more information on the operation and advantages of the FEMMS instruction, see Chapter 4, "3D Technology" on page 81.

**Use the FEMMS instruction at the beginning of an MMX or 3D routine**—While the FEMMS instruction is not necessary for correct program functionality at the beginning of MMX or 3D routines, its usage reduces the clock-cycle count penalty when entering such routines from preceding x87 code. If no switch occurs, the FEMMS takes 3 clock cycles to execute. If a switch is necessary, FEMMS reduces the clock cycles required by over half.

Practice the following general rules when using MMX or 3D code mixed with x87 code:

- Always use the FEMMS instruction (instead of EMMS) at the end of an MMX or 3D routine when x87 instructions or unknown code follows.
- Use the FEMMS instruction at the beginning of an MMX or 3D routine that is preceded by x87 instructions or unknown code. FEMMS serves to reduce any switch penalty.
- Group or partition MMX or 3D code separate from x87 code to minimize the frequency of switching between MMX or 3D operations and x87 operations.

Use the new 3D instruction PAVGUSB instruction for MPEG-2 motion compensation—In DVD decoding, motion compensation performs a lot of byte averaging between and within macroblocks. The PAVGUSB instruction helps speed up these operations. In addition, PAVGUSB can free up some registers and make unrolling the averaging loops possible.

## **B** Code Optimization

The following code fragment uses original MMX code to perform averaging between the source macroblock and destination macroblock:

```
mov     esi, DWORD PTR Src_MB
mov     edi, DWORD PTR Dst_MB
mov     edx, DWORD PTR SrcStride
mov     ebx, DWORD PTR DstStride
movq    mm7, QWORD PTR [ConstFEFE]
movq    mm6, QWORD PTR [Const0101]
mov     ecx, 16

L1:
movq    mm0, [esi]           ;mm0=qword1
movq    mm1, [edi]           ;mm1=qword3
movq    mm2, mm0
movq    mm3, mm1
pand    mm2, mm6
pand    mm3, mm6
pand    mm0, mm7             ;mm0 = qword1 & 0xfefefefe
pand    mm1, mm7             ;mm1 = qword3 & 0xfefefefe
por     mm2, mm3             ;calculate adjustment
psrlq   mm0, 1               ;mm0 = (qword1 & 0xfefefefe)/2
psrlq   mm1, 1               ;mm1 = (qword3 & 0xfefefefe)/2
pand    mm2, mm6
paddb   mm0, mm1             ;mm0 = qw1/2 + qw3/2 w/o adjust-
                               ;      ment
paddb   mm0, mm2             ;add 1sb adjustment
movq    [edi], mm0
movq    mm4, [esi+8]         ;mm4=qword2
movq    mm5, [edi+8]         ;mm5=qword4
movq    mm2, mm4
movq    mm3, mm5
pand    mm2, mm6
pand    mm3, mm6
pand    mm4, mm7             ;mm0 = qword2 & 0xfefefefe
pand    mm5, mm7             ;mm1 = qword4 & 0xfefefefe
por     mm2, mm3             ;calculate adjustment
psrlq   mm4, 1               ;mm0 = (qword2 & 0xfefefefe)/2
psrlq   mm5, 1               ;mm1 = (qword4 & 0xfefefefe)/2
pand    mm2, mm6
paddb   mm4, mm5             ;mm0 = qw2/2 + qw4/2 w/o adjust-
                               ;      ment
paddb   mm4, mm2             ;add 1sb adjustment
movq    [edi+8], mm4

add     esi, edx
add     edi, ebx
loop    L1
```

The following code fragment uses the 3D PAVGUSB instruction to perform averaging between the source macroblock and destination macroblock:

```

mov     eax, DWORD PTR Src_MB
mov     edi, DWORD PTR Dst_MB
mov     edx, DWORD PTR SrcStride
mov     ebx, DWORD PTR DstStride
mov     ecx, 16

L1:
movq    mm0, [eax]           ;mm0=qword1
movq    mm1, [eax+8]         ;mm1=qword2
pavgusb mm0, [edi]           ;(qw1+qw3)/2 with adjustment
pavgusb mm1, [edi+8]         ;(qw2+qw4)/2 with adjustment
add     eax, edx
movq    [edi], mm0
movq    [edi+8], mm1
add     edi, ebx
loop    L1

```

### 3D Matrix Multiplication Optimization Example

The code samples starting on page 488 contain both a non-optimized and an optimized sample of a 4x4 matrix multiplied by a 4x1 vector. This type of code is often used in 3D graphics for geometry transformation. This routine serves to translate, scale, rotate, and apply perspective to 3D coordinates represented in homogeneous coordinates. The code samples contain many addition and multiplication instructions that can now be implemented in any one of three ways. For high-end, 3D graphic programs, x87 FPU instructions supply only moderate performance, are not superscalar, and cannot be efficiently intermixed with MMX and 3D instructions. Integer instructions and MMX instructions, while fast and superscalar, do not have the accuracy and dynamic range that is required for these programs. Therefore, the 3D instructions, providing the benefit of packed, floating-point data precision and parallel execution, can be used in order to write software that outperforms standard floating-point code and has no switching overhead when intermixed with MMX code. The following two code samples illustrate non-optimized and optimized code. A description of the steps a programmer should take when optimizing code for the AMD-K6 3D processor starts on page 493.

# B Code Optimization

## Non-Optimized Code Sample:

```

;-----
; void Transform4x4(Vertex *firstv, int cnt, const Matrix *m)
;
; NON OPTIMIZED VERSION
;
; Full 4x4 matrix transform of an array of cnt vertices starting from the
; vertex pointed to by firstv, using the transform matrix pointed to by m.
;
; Each vertex data structure is assumed to occupy 128 bytes, 16 bytes of
; which contains the vertex coordinates to be transformed.
;
;   new_x = x*m[0][0] + y*m[0][1] + z*m[0][2] + w*m[0][3];
;   new_y = x*m[1][0] + y*m[1][1] + z*m[1][2] + w*m[1][3];
;   new_z = x*m[2][0] + y*m[2][1] + z*m[2][2] + w*m[2][3];
;   new_w = x*m[3][0] + y*m[3][1] + z*m[3][2] + w*m[3][3];
;-----
Vrtx_X equ 0h
Vrtx_Y equ 4h
Vrtx_Z equ 8h
Vrtx_W equ 0ch
Mat_00 equ 0h
Mat_01 equ 4h
Mat_02 equ 8h
Mat_03 equ 0ch
Mat_10 equ 10h
Mat_11 equ 14h
Mat_12 equ 18h
Mat_13 equ 1ch
Mat_20 equ 20h
Mat_21 equ 24h
Mat_22 equ 28h
Mat_23 equ 2ch
Mat_30 equ 30h
Mat_31 equ 34h
Mat_32 equ 38h
Mat_33 equ 3ch

;EAX = m      ptr to transform matrix
;EBX = firstv  ptr to first vertex to be transformed
;EDX = lastv   ptr to last vertex to be transformed

```

*Comments appear after the code lines.*

```

TransformLoop:
                                ;All multiplies for XResult:
movq  mm0, QWORD PTR [ebx + Vrtx_X] ;mm0 = y * x
movq  mm2, mm0                  ;copy vector
                                ;Right in the beginning there is a dependency for mm0, which stalls the second movq 2 clock cycles, even though
                                ;both instructions are short-decodeable and decode together as an instruction pair.

```

```
pfmul mm0, QWORD PTR [eax + Mat_00] ;mm0 = y*a21 | x*a11
```

*The PFMUL instruction leads to another dependency, but because of the previous stall, the PFMUL instruction executes when Mat\_00 loads from memory. The PFMUL instruction translates to an 3D ALU and a Load unit operation.*

```
movq mm1, QWORD PTR [ebx + Vrtx_Z] ;mm1 = w | z
```

*The MOVQ instruction decodes with the previous pfmul but there is now a resource constraint, with both instructions trying to use the Load unit. This contention causes one of the instructions to stall an extra cycle.*

```
movq mm3, mm1 ;copy vector
```

*Another stall while waiting for mm1.*

```
pfmul mm1, QWORD PTR [eax + Mat_20] ;mm1 = w*a41 | z*a31
```

*Same as the previous PFMUL instruction. Note that tasks in this code line are serialized, with no opportunity for overlap of execution resources. Even if the instructions short decode in pairs, other constraints are causing stalls. In addition, a scheduler stall occurs when an instruction cannot retire off the bottom of the scheduler because dependency and resource stalls have delayed the instruction too many cycles.*

```
movq mm4, mm2 ;All multiplies for YResult:
;copy vector
```

```
pfmul mm2, QWORD PTR [eax + Mat_01] ;mm2 = y*a22 | x*a12
```

*These instructions are paired. The PFMUL instructions decode to a Load unit operation followed by an 3D Multiply unit operation.*

```
movq mm5, mm3 ;copy vector
```

```
pfmul mm3, QWORD PTR [eax + Mat_21] ;mm3 = w*a42 | z*a32
```

*These instructions are paired. Same comments as before.*

```
movq mm6, mm4 ;All multiplies for ZResult:
;copy vector
```

```
pfmul mm4, QWORD PTR [eax + Mat_02] ;mm4 = y*a23 | x*a13
```

*These instructions are paired. Same comments as before.*

```
movq mm7, mm5 ;copy vector
```

```
pfmul mm5, QWORD PTR [eax + Mat_22] ;mm5 = w*a43 | z*a33
```

*These instructions are paired. Same comments as before.*

```
pfmul mm6, QWORD PTR [eax + Mat_03] ;All multiplies for WResult:
;mm6 = y*a24 | x*a14
```

```
pfmul mm7, QWORD PTR [eax + Mat_23] ;mm7 = w*a44 | z*a34
```

*These instructions are paired. However, this pair causes a conflict for both the Load unit and the 3D Multiplier resources, which stalls one instruction in the scheduler for a clock cycle. The instructions execute in a staggered fashion. The goal for short-decodeable pairs is simultaneous execution.*

```
pfadd mm0, mm1 ;All first sums:
; of XResult
;mm0 = w*a41 + y*a21 | z*a31 + x*a11
```

```
pfadd mm2, mm3 ; of YResult
;mm2 = w*a42 + y*a22 | z*a32 + x*a12
```

*These instructions are paired. However, this pair causes a conflict for the 3D ALU, which delay one instruction.*

```
pfadd mm4, mm5 ; of ZResult
;mm4 = w*a43 + y*a23 | z*a33 + x*a13
```

## B Code Optimization

```

                                ; of WResult
pfadd mm6, mm7                ;mm6 = w*a44 + y*a24 | z*a34 + x*a14
    These instructions are paired, but there is a conflict for the 3D ALU and with one of the PFADD instructions from the
    previous pair that was delayed one cycle. These dual-decodeable operations serialize execution, eventually stalling
    the scheduler because RISC86 instructions can no longer retire.

                                ;All final sums:
pfacc mm0, mm0                ; of XResult
pfacc mm2, mm2                ; of YResult
    These instructions are paired, but there is a conflict for the 3D ALU. See the comments above.

pfacc mm4, mm4                ; of ZResult
pfacc mm6, mm6                ; of WResult
    These instructions are paired, but there is a conflict for the 3D ALU. See the comments above.

                                ;All result stores:
movd  DWORD PTR [ebx + Vrtx_X], mm0 ; of XResult
movd  DWORD PTR [ebx + Vrtx_Y], mm2 ; of YResult
    These instructions are paired, but there is a conflict for the Store unit.

movd  DWORD PTR [ebx + Vrtx_Z], mm4 ; of ZResult
movd  DWORD PTR [ebx + Vrtx_W], mm6 ; of WResult
    These instructions are paired, but there is a conflict for the Store Unit as well as the delayed store operation from
    the previous instruction pair.

add  ebx, Vertex_Stride        ;Advance to next vertex
cmp  ebx, edx                  ;Compare with ptr to last vertex
    These instructions are paired, but a dependency on ebx value delays the second instruction by one cycle.

jbe  TransformLoop            ;If not done yet

```

### Optimized Code Sample:

```

; void Transform4x4(Vertex *firstv, int cnt, const Matrix *m)
;
; OPTIMIZED VERSION
;
; Full 4x4 matrix transform of an array of cnt vertices starting from the
; vertex pointed to by firstv, using the transform matrix pointed to by m.
;
; Each vertex data structure is assumed to occupy 128 bytes, 16 bytes of
; which contains the vertex coordinates to be transformed.
;
; new_x = x*m[0][0] + y*m[0][1] + z*m[0][2] + w*m[0][3];
; new_y = x*m[1][0] + y*m[1][1] + z*m[1][2] + w*m[1][3];
; new_z = x*m[2][0] + y*m[2][1] + z*m[2][2] + w*m[2][3];
; new_w = x*m[3][0] + y*m[3][1] + z*m[3][2] + w*m[3][3];
;
-----
Vrtx_X equ 0h
Vrtx_Y equ 4h
Vrtx_Z equ 8h

```

```

Vrtx_W equ 0ch
Mat_00 equ 0h
Mat_01 equ 4h
Mat_02 equ 8h
Mat_03 equ 0ch
Mat_10 equ 10h
Mat_11 equ 14h
Mat_12 equ 18h
Mat_13 equ 1ch
Mat_20 equ 20h
Mat_21 equ 24h
Mat_22 equ 28h
Mat_23 equ 2ch
Mat_30 equ 30h
Mat_31 equ 34h
Mat_32 equ 38h
Mat_33 equ 3ch

```

```

;EAX = m      ptr to transform matrix
;EBX = first_v ptr to first vertex to be transformed
;ECX = cnt     count of vertices to be transformed

```

*The code begins here, but this section is not in the loop. The initial loads conflict and stall waiting to load the first vertex values and the first four values from the matrix. However, once the loop begins, this code runs efficiently. Note that most of these x86 instructions are four bytes long, which helps to make them short decodable.*

```

                                ;Load first vertex:
movq mm6, DWORD PTR [ebx]      ;mm6 = y | x
movq mm7, DWORD PTR [ebx + Vrtx_Z] ;mm7 = w | z

```

*These instructions decode together, but cause a conflict for the Load unit.*

```

                                ;Start load of matrix:
movq mm0, DWORD PTR [eax + Mat_00] ;mm0 = m01 | m00
movq mm1, DWORD PTR [eax + Mat_20] ;mm1 = m03 | m02

```

*Decode together, but conflict for the Load Unit.*

```

TransformLoop:
    prefetchw [ebx + 128]      ;Prefetch next vertex

```

*The PREFETCHW instruction is a vector decode and takes 2 cycles. However, this instruction increases efficiency because it begins the preload of the L1 data cache with the next vertex. A vertex is four dwords or 1/2 a cache line. However, the 'stride' or distance from one vertex data structure to the next within the vertex array, in this example, is 128 bytes, which means that each vertex is in a separate cache line. It is assumed that vertex data starts on cache line boundaries. From this point forward, the x86 instructions form instruction pairs that both decode into one Opquad. An Opquad is one line in the instruction scheduler that is composed of four RISC86 operations.*

```

movq mm2, DWORD PTR [eax + Mat_01] ;mm2 = m11 | m10

```

*This MOVQ instruction continues to fill in the matrix. Separating the matrix load from the multiply instruction avoids serializing the load and multiply, which can lead to a stall of the scheduler. The load takes 2-3 cycles to execute and the multiply takes 2 cycles to execute. Including the operand fetch stage almost fills the six-stage length of the scheduler.*

```

pfmul mm0, mm6                ;mm0 = y*m01 | x*m00

```

*This PFMUL instruction is paired with the MOVQ instruction. These two instructions use different resources (Load*



## B Code Optimization

Unit and 3D ALU, respectively). There are no resource conflicts, no dependencies (mm0 should be loaded from three cycles earlier), and the instructions execute together.

```
movq mm3, DWORD PTR [eax + Mat_21] ;mm3 = m13 | m12
pfmul mm1, mm7 ;mm1 = w*m03 | z*m02
```

Same comments as previous instruction pair.

```
movq mm4, DWORD PTR [eax + Mat_02] ;mm4 = m21 | m20
pfmul mm2, mm6 ;mm2 = y*m11 | x*m10
```

Same comments as previous instruction pair, except the load mm2 was started two cycles earlier and should be forwarded from the Load unit to the 3D ALU just-in-time.

```
movq mm5, DWORD PTR [eax + Mat_22] ;mm5 = m23 | m22
pfmul mm3, mm7 ;mm3 = w*m13 | z*m12
```

In this pair of instructions, the last free register is loaded for now (mm5). Because there are only eight MMX registers, the registers must be reused and then reloaded with the matrix values for the next vertex calculation.

```
pfadd mm0, mm1 ;First sum of XResult:
pfmul mm4, mm6 ;mm0 = w*m03 + y*m01 | z*m02 + x*m00
;mm4 = y*m21 | x*m20
```

These two 3D instructions can be paired because the 3D ALU and multiplier are separate units and both have access to the issue buses for the register X and register Y execution pipelines. Note that at this time the processor is operating on eight single-precision, floating-point values (packed into four mmx registers) and the processor produces four single-precision values (in two mmx registers).

```
pfadd mm2, mm3 ;First sum of YResult:
;mm2 = w*m13 + y*m11 | z*m12 + x*m10
```

The mm3 operand is forwarded from the 3D multiplier output.

```
movq mm1, DWORD PTR [eax + Mat_03] ;mm1 = m31 | m30
```

The previous two instructions are paired. The MOVQ instruction moves in the first pair of the remaining four matrix values.

```
pfmul mm5, mm7 ;mm5 = w*m23 | z*m22
```

The mm5 operand is forwarded from the Load unit and the mm7 operand is forwarded from the 3D multiplier.

```
movq mm3, DWORD PTR [eax + Mat_23] ;mm3 = m33 | m32
```

The previous two instructions are paired. The MOVQ instruction moves in the last pair of matrix values.

```
add ebx, Vertex_Stride ;Advance to next vertex
```

The pointer to the next vertex is updated. In this example, Vertex\_Stride = 128.

```
pfmul mm1, mm6 ;mm1 = y*m31 | x*m30
```

The previous two instructions are paired.

```
pfacc mm0, mm2 ;Final sum of XResult and YResult:
;mm0 = YRes | XRes
```

The first pair of vertex values are complete and can be stored two clock cycles later (the 3D accumulate has a two-cycle execution latency, as do all 3D ALU and Multiply instructions).

```
pfmul mm3, mm7 ;mm3 = w*m33 | z*m32
```

The previous two instructions are paired and use the 3D ALU and Multiplier units simultaneously.

```

                                ;First sum of ZResult
pfadd mm4, mm5                  ;mm4 = w*m23 + y*m21 | z*m22 + x*m20
    Continuing the goal of spreading out dependencies, this instruction is two cycles after the mm5 calculation.

                                ;Load next vertex
movq mm6, DWORD PTR [ebx + Vrtx_X] ;mm6 = y | x
    The previous two instructions are paired. This MOVQ instruction begins to load the next vertex, which the PREFETCH instruction has been preloading into the L1 data cache.

                                ;First sum of WResult:
pfadd mm1, mm3                  ;mm1 = w*m33 + y*m31 | z*m32 + x*m30
                                ;Load next vertex
movq mm7, DWORD PTR [ebx + Vrtx_Z] ;mm7 = w | z
    The previous two instructions are paired. The second part of the new vertex is loaded.

movq DWORD PTR [ebx - 128 + Vrtx_X], mm0 ;Store XResult and YResult
                                ;Start next iteration
movq mm0, DWORD PTR [eax + Mat_00] ;mm0 = m01 | m00
    The previous two instructions are paired and can complete simultaneously because the AMD-K6 3D processor has separate Load and Store units. Unfortunately, all the matrix values must be reloaded with each iteration because there are not enough registers to hold the vertices, the full matrix, and intermediate values.

                                ;Final sum of ZResult and WResult:
pfacc mm4, mm1                  ;mm4 = WRes | ZRes
                                ;Start next iteration
movq mm1, DWORD PTR [eax + Mat_20] ;mm1 = m03 | m02
    The previous two instructions are paired.

movq DWORD PTR [ebx - 128 + Vrtx_Z], mm4 ;Store ZResult and WResult
    Fortunately, the Store unit can accept data up to two cycles later without a penalty because there are no calculations left to hide the execution latency of the last accumulate instruction. Therefore, this store is not delayed.

loop Transformloop              ;If not done yet, go to beginning of the loop.
    The previous two instructions are short decodeable and paired. Note that on the processor, the LOOP instruction executes in the same amount of time as the CMP and JBE instructions in the non-optimized example. However, the LOOP instruction, being only one instruction instead of two, is more efficient.

```

**Programming Steps**

The following descriptions review and expand on the steps taken to arrive at the optimized code example:

**Schedule code into pairs of short-decodeable x86 instructions that correspond to the expected decode pairing**—Each short-decodeable pair of instructions decodes into four RISC86 operations that form a set of four Op entries in the instruction scheduler. This set of entries moves down the scheduler and eventually retires from the bottom of the scheduler buffer. The scheduler buffer can hold a total of six sets of entries (which represents a total of 24 Op entries). Under ideal conditions of uninterrupted decode and execution (no stalls), these entries also correspond to clock cycles through the scheduler and

## **B** *Code Optimization*

execution pipes of the AMD-K6 3D processor. Consequently, the programmer should schedule dependent instructions apart from each other, in different decode pairs, based on the execution latencies of the corresponding RISC86 operations. It is cleanest and simplest to use only MOVQ and MOVD instructions for memory loads and stores, and use register-to-register instructions for computations. In addition, this technique has the benefit of minimizing or avoiding instruction scheduling delays due to long-latency instructions (such as those with a memory load followed by a two-cycle register operation), not completing in time and, therefore, not being ready to commit results when the entry containing the associated RISC86 operations reaches the bottom of the scheduler buffer. This situation can lead to a stall when no new RISC86 operations can be placed in the scheduler until an entry is available.

**Interleave independent sequences of instructions (subject to register allocation constraints) to fill each and every decode slot**—To the extent that this is achieved while maintaining the proper minimum distances between dependent operations and respecting execution resource constraints, optimal decode pairing and instruction execution without delays or stalls is very likely to be achieved.

**Use separate moves from memory and register-to-register multiplies, instead of register-to-register copies and multiplies from memory**—This technique allows easy explicit and optimal scheduling of memory loads and dependent register operations, spaced at least two decode pairs apart and corresponding to the two-cycle load execution latency. While this technique generally applies to all MMX and 3D instructions, particularly avoid the use of the memory form of instructions with two-cycle execution latencies (for example, all 3D instructions). In other words, optimal performance is best and most easily achieved using a RISC coding style (despite the extra MOVD/MOVQ instructions).

**Schedule instructions apart that use the same execution resources**—For example, multiplies should be spread apart. The programmer should put at least one decode slot between multiplies. Similarly, adds and accumulates, memory loads, and memory stores should be spread apart.

**Use the pipelining ability of accumulate instructions to perform two independent accumulates and to pair the resultant values together as a 64-bit result—**This technique allows the use of fewer MOVQ instructions instead of a greater number of MOVD instructions. Overall, four accumulate and four MOVD instructions get replaced with two and two. In some situations, where scalar register results are naturally produced and are then stored out to memory via a series of MOVDs, it may be preferable to reduce the number of store operations through use of PUNPCKLDQ instructions followed by MOVQ instructions. Often this optimization may not be worthwhile or favorable, particularly given the extra latency introduced by the PUNPCKLDQ operations and possibly by memory alignment issues for the MOVQ instructions. Typically, it is best to spend the overhead to pack initial scalar operands together when first read from memory (using MOVD instructions), followed by vector computations and MOVQs back to memory.

**Separate the first and second stores by at least two or three decode slots (in other words, by one intervening decode pair) within a series of two or more stores to a cache line recently brought in and not yet written to—**This technique is in contrast to the second and following stores, which can be in adjacent decode pairs. This technique allows an extra cycle for the initial MESI-state change to the cache line (from Exclusive to Shared).

**Schedule the ADD/CMP/JCC instructions apart (or at least the ADD and CMP instructions)—**This scheduling is primarily desirable when the ADD and/or CMP instructions reference a memory operand and are, therefore, subject to the latency of the load operation. In such cases, either the ADD/CMP instruction should be scheduled apart from (and ahead of) the JCC instructions, or a separate MOV instruction, scheduled earlier, should be used to fetch the memory operand. An alternative and desirable solution in some cases is to replace these instructions with the LOOP instruction (along with corresponding setup and usage of the ECX register before and within the loop).

**Take advantage of the PREFETCH instruction—**In the optimized code example, each vertex occupies a different cache line (the stride between vertices being 32 bytes or greater). Consequently, one cache miss and associated 32 byte line fill occurs per loop iteration. To maximize overlap of the cache fill

## **B** *Code Optimization*

from L2 cache or main memory, use the PREFETCH instruction to start the fill before starting to process the current vertex (which is already in the cache, having been prefetched at the beginning of processing of the last vertex). Specify the address elements of the next vertex that will be accessed first. In addition, schedule the loads of the next vertex's data elements away from the prefetch instruction. Doing so ensures that the load data (which will be the first data of the cache line to be fetched) has been received and is available for forwarding to these loads while the rest of the fill proceeds to completion.

**Move the first few MOVQs around to the bottom of the loop—**Typically, the first instructions after the prefetch instruction would be a series of MOVQs to get the first vertex and matrix elements to operate on, without any other available independent operations to fill out these first couple of decode pairs. Similarly, near the bottom of the loop, as the last computations are performed, there would also be some partially-filled decode slots. To fix both of these problems, move the first few vertex and matrix element MOVQ instructions from the bottom of the loop into the empty slots (as well as duplicating these MOVQs in the setup code before the start of the loop).

**Pay attention to the alignment of instructions relative to 32-byte cache line boundaries—**The code samples do not show the actual memory alignment of instructions and, therefore, whether the decode of any instructions may be impacted by end-of-cache-line degraded predecode. These code examples require a suitable starting alignment (relative to a 32-byte address boundary). There also exists the possibility that there is no starting alignment for which all instructions can be successfully predecoded. In such cases, adjustments to the code (such as padding with one-byte or multiple-byte NOPs, instruction rearrangement, or different instruction selections) may be warranted. In the case of 3D instructions, which can still be hardware decoded as a single long decode, the best alternative may sometimes be to do nothing.

**Avoid certain address modes with MMX and 3D instructions that inhibit instruction predecode—**As discussed earlier in the section, the [ESI] modR/M address mode (without any displacement bytes or index register) inhibits successful instruction predecode and should be avoided. In addition, for

MMX and 3D instructions, address modes that use a sib byte with `mod=00b` in the `modR/M` byte should be avoided. These cases consist of `[base + (scaled)index]` and `[disp32 + (scaled)index]` address modes.

## Division

The 3D instructions can be used to compute a very fast, highly accurate reciprocal or quotient.

Consider the quotient  $q = a/b$ . An (on-chip) ROM-based table lookup can be used to quickly produce a 14-to-15-bit precision approximation of  $1/b$ . (Using just one 2-cycle latency `PFRCP` instruction). A full 24-bit precision reciprocal can then be quickly computed from this approximation using a Newton Raphson algorithm.

The general Newton-Raphson recurrence for the reciprocal is as follows:

$$X_{i+1} = X_i \cdot (2 - b \cdot X_i)$$

Given that the initial approximation  $X_0$  is accurate to at least 14 bits, and that a full IEEE single-precision mantissa contains 24 bits, just one Newton-Raphson iteration is required. The following sequence shows the 3D instructions that produce the initial reciprocal approximation, compute the full precision reciprocal from the approximation, and finally, complete the desired divide of  $a/b$ .

$$X_0 = \text{PFRCP}(b)$$

$$X_1 = \text{PFRCPIT1}(b, X_0)$$

$$X_2 = \text{PFRCPIT2}(X_1, X_0)$$

$$q = \text{PFMUL}(a, X_2)$$

The 24-bit final reciprocal value is  $X_2$ . In the AMD-K6 3D processor 3D implementation, the estimate contains the correct round-to-nearest value for approximately 99% of all arguments. The remaining arguments differ from the correct round-to-nearest value for the reciprocal by 1 unit-in-the-last-place (ulp). The quotient is formed in the last step by multiplying the reciprocal by the dividend  $a$ .

## B Code Optimization

### Optimized 15-Bit Precision Divide

This divide operation executes with a total latency of 4 cycles, assuming that the programmer is able to hide the latency of the first MOVD/MOVQ instructions within preceding code.

```
MOVD    MM0, [mem]      ; 0|w
PFRCP   MM0, MM0        ; 1/w|1/w
MOVQ    MM2, [mem]      ; y|x
PFMUL   MM2, MM0        ; y/w|x/w
```

### Optimized Full 24-Bit Precision Divide

This divide operation executes with a total latency of 8 cycles, assuming that the programmer is able to hide the latency of the first MOVD/MOVQ instructions within preceding code.

```
MOVD    MM0, [mem]      ; 0|w
PFRCP   MM1, MM0        ; 1/w|1/w
PFRPIT1 MM0; MM1        ;
MOVQ    MM2, [mem]      ; y|x
PFRCPIT2 MM0, MM1       ; 1/w|1/w
PFMUL   MM2, MM0        ; y/w|x/w
```

### Pipelined Pair of 24-Bit Precision Divides

This divide operation executes with a total latency of 8 cycles, assuming that the programmer is able to hide the latency of the first MOVD/MOVQ instructions within preceding code.

```
MOVD    MM1, [mem]      ; 0|w0
MOVD    MM2, [mem+4]    ; 0|w1
PFRCP   MM1, MM1        ; 1/w0|1/w0
MOVQ    MM0, [mem]      ;
PFRCP   MM2, MM2        ; 1/w1|1/w1
PUNPCKLDQ MM1, MM2     ; 1/w1|1/w0
PFRCPIT1 MM0, MM1       ;
MOVQ    MM2, [mem]      ; y|x
PFRCPIT2 MM0, MM1       ; 1/w1|1/w0
PFMUL   MM2, MM0        ; y/w1|x/w0
```

## Square Root and Reciprocal Square Root

The 3D instructions can also be used to compute a reciprocal square root or square root with high performance. The general Newton-Raphson reciprocal square root recurrence is:

$$X_{i+1} = 1/2 \cdot X_i \cdot (3 - b \cdot X_i^2)$$

To reduce the number of iterations,  $X_0$  is an initial approximation read from a table. The 3D reciprocal square root approximation is accurate to at least 15 bits. Accordingly, to obtain a single-precision 24-bit reciprocal square root of an

input operand *b*, one Newton-Raphson iteration is required, using the following sequence of 3D instructions:

```

X0 = PFRSQRT(b)
X1 = PFMUL(X0, X0)
X2 = PFRSQIT1(b, X1)
X3 = PFRCPIT2(X2, X0)
X4 = PFMUL(b, X3)

```

The 24-bit final reciprocal square root value is *X*<sub>3</sub>. In the AMD-K6 3D processor 3D implementation, the estimate contains the correct round-to-nearest value for approximately 87% of all arguments. The remaining arguments differ from the correct round-to-nearest value by 1 ulp. The square root (*X*<sub>4</sub>) is formed in the last step by multiplying by the input operand *b*.

#### Optimized 15-Bit Precision Square Root

This square root operation can be executed in only 4 cycles, assuming a programmer is able to hide the latency of the first MOVD instruction within previous code. The reciprocal square root operation requires two less cycles than the square root operation.

```

MOVD    MM0, [mem]      ; 0|a
PFRSQRT MM1, MM0        ; 1/sqrt(a)|1/sqrt(a)
PFMUL   MM0, MM1        ; sqrt(a)|sqrt(a)

```

#### Optimized 24-Bit Precision Square Root

This square root operation can be executed in only 10 cycles, assuming a programmer is able to hide the latency of the first MOVD instruction within previous code. The reciprocal square root operation requires two less cycles than the square root operation.

```

MOVD    MM0, [mem]      ; 0|a
PFRSQRT MM1, MM0        ; 1/sqrt(a)|1/sqrt(a)
MOVQ    MM2, MM1        ;
PFMUL   MM1, MM1        ;
PFRSQIT1 MM1, MM0        ;
PFRCPIT2 MM1, MM2        ; 1/sqrt(a)|1/sqrt(a)
PFMUL   MM0, MM1        ; sqrt(a)|sqrt(a)

```



# **B** *Code Optimization*

---

## **x87 Floating-Point Coding Optimizations**

This section describes x87 floating-point code optimization techniques specific to the AMD-K6 3D processor.

**For optimal floating-point performance**—Wherever possible, use the packed single-precision, floating-point capability of 3D technology instead of the single-precision, double-precision, and extended-precision floating-point capabilities of the x87 floating-point unit. The 3D units are fully pipelined, allow vectorized optimizations, are not stack based, and provide faster inverse, square root, and inverse square root calculations.

**Avoid vector decoded floating-point instructions**—Most floating-point instructions are short decodeable. A few of the less common instructions are vector decoded. In addition, if a short decodeable instruction straddles a cache line, it becomes vector decoded. This adds unnecessary overhead that can be avoided by inserting NOPs in strategic locations within the code.

**Pair floating-point with short-decodeable instructions**—Most floating-point instructions (also known as ESC instructions) are short-decodeable and are limited to the first decoder. The short-decodeable floating-point instructions can be paired with other short-decodeable instructions. This technique requires that floating-point instructions be arranged as the first of a pair of short-decodeable instructions.

**Avoid FXCH usage**—Pairing FXCH with other floating-point instructions does not increase performance.

**Minimize switching between MMX or 3D instructions and FPU instructions**—Because the MMX/3D registers are mapped onto the floating-point register stack, the EMMS or FEMMS instruction must be executed after MMX or 3D code and prior to the use of the floating-point unit. Group or partition MMX and 3D code away from FPU code so that the use of the EMMS or FEMMS instructions is minimized. In addition, the actual penalty or switch overhead from the use of the EMMS or FEMMS instructions occurs not at the time of their execution, but when and if the first floating-point instruction is encountered.

**Avoid using MMX/3D registers (and MOVQ instructions) to move blocks of double-precision floating-point data in memory**—Although using 64-bit MOVQ instructions to move floating-point data appears fast, using MMX/3D registers requires the use of the EMMS or FEMMS instruction and incurs switch overhead when switching between these MMX or 3D instructions and surrounding floating-point instructions.

**Exploit parallel execution of integer and floating-point multiplies**—The AMD-K6 3D processor allows simultaneous integer and floating-point multiplies using separate, low-latency multipliers.

**Do not split floating-point instructions with integer instructions**—No penalty is incurred when using arithmetic or comparison floating-point instructions that use integer operands, such as the FIADD instruction or FICOM instruction. Splitting these instructions into discrete load and floating-point instructions decreases performance.

**Replace FDIV instructions with FMUL where possible**—The FMUL instruction latency is much less than the FDIV instruction. If possible, replace floating-point divisions with floating-point multiplication of the reciprocal.

**Use integer instructions to move floating-point data**—A floating-point load and store instruction pair requires a minimum of four cycles to complete (two-cycle latency for each instruction). The AMD-K6 3D processor can perform one integer load and one store per cycle. Therefore, moving single-precision data requires one cycle, moving double-precision data requires two cycles, and moving extended-precision data only requires three cycles when using integer loads and stores. The following example shows how to translate the C-style code when moving double-precision floating-point data:

```
double temp1, temp2;
temp2 = temp1;
```

```
FLDQWORD PTR [temp1]:  Use:  MOV  EAX, [temp1];
FSTP QWORD PTR [temp2]: MOV   [temp2], EAX;
                       MOV   EAX, [temp1+4];
                       MOV   [temp2+4], EAX;
```

## **B** Code Optimization

**Scheduling of floating-point instructions is unnecessary—**The AMD-K6 3D processor has a low-latency, non-pipelined floating-point execution unit.

**Use load-execute floating-point instructions—**The use of a load-execute instruction (such as, `FADD DWORD PTR [mem]`) is preferable to the use of a load floating-point instruction followed by a floating-point `reg, reg` instruction. For the AMD-K6 3D processor, load-execute arithmetic and compare instructions are identical in throughput to floating-point `reg, reg` instructions. Because common floating-point instructions execute in two cycles each and the floating-point unit is not pipelined, code executes more efficiently if the minimum possible number of floating-point instructions are generated.

### **Floating-Point Code Sample**

The following code sample uses three of the most important rules to optimize this matrix multiply routine. The first rule used is avoidance of the `[ESI]` addressing mode. The routine forces this code to be `[ESI+0]`. The second rule is the insertion of `NOP`s to avoid cache-line straddles. The third rule used is avoidance of vector decoded instructions.

```
MATMUL    MACRO
    db     0d9h, 046h, 00h ;; FLD DWORD PTR [ESI+00] ;; x
    FMUL   DWORD PTR [EBX] ;; a11*x
    FLD    DWORD PTR [ESI+4] ;; y
    FMUL   DWORD PTR [EBX+4] ;; a21*y
    FLD    DWORD PTR [ESI+8] ;; z
    FMUL   DWORD PTR [EBX+8] ;; a31*z
    FLD    DWORD PTR [ESI+12] ;; w
    FMUL   DWORD PTR [EBX+12] ;; a41*w
    FADDP  ST(3), ST ;; a41*w+a31*z
    FADDP  ST(2), ST ;; a41*w+a31*z+a21*y
    FADDP  ST(1), ST ;; a41*w+a31*z+a21*y+a11*x
    FSTP   DWORD PTR [EDI] ;; store rx
    NOP    ;; make sure it does not
            ;; straddle across a cache line
    db     0d9h, 046h, 00h ;; FLD DWORD PTR [ESI+00] ;; x
    FMUL   DWORD PTR [EBX+16] ;; a12*x
    FLD    DWORD PTR [ESI+4] ;; y
    FMUL   DWORD PTR [EBX+20] ;; a22*y
    FLD    DWORD PTR [ESI+8] ;; z
    NOP    ;; make sure it does not
            ;; straddle across a cache line
    FMUL   DWORD PTR [EBX+24] ;; a32*z
    FLD    DWORD PTR [ESI+12] ;; w
    FMUL   DWORD PTR [EBX+28] ;; a42*w
    FADDP  ST(3), ST ;; a42*w+a32*z
    FADDP  ST(2), ST ;; a42*w+a32*z+a22*y
```

```

FADDP ST(1), ST;; a42*w+a32*z+a22*y+a12*x
NOP          ;; make sure it does not
              ;; straddle across a cache line
FSTP  DWORD PTR [EDI+4];; store ry
db    0d9h, 046h, 00h;; FLD DWORD PTR [ESI+00] ;; x
FMUL  DWORD PTR [EBX+32];; a13*x
FLD   DWORD PTR [ESI+4];; y
FMUL  DWORD PTR [EBX+36];; a23*y
NOP          ;; make sure it does not
              ;; straddle across a cache line
FLD   DWORD PTR [ESI+8];; z
FMUL  DWORD PTR [EBX+40];; a33*z
FLD   DWORD PTR [ESI+12];; w
FMUL  DWORD PTR [EBX+44];; a43*w
FADDP ST(3), ST;; a43*w+a33*z
FADDP ST(2), ST;; a43*w+a33*z+a23*y
FADDP ST(1), ST;; a43*w+a33*z+a23*y+a13*x
FSTP  DWORD PTR [EDI+8];; store rz
db    0d9h, 046h, 00h;; FLD DWORD PTR [ESI+00] ;; x
FMUL  DWORD PTR [EBX+48];; a14*x
FLD   DWORD PTR [ESI+4];; y
FMUL  DWORD PTR [EBX+52];; a24*y
FLD   DWORD PTR [ESI+8];; z
FMUL  DWORD PTR [EBX+56];; a34*z
FLD   DWORD PTR [ESI+12];; w
FMUL  DWORD PTR [EBX+60];; a44*w
FADDPST(3), ST;; a44*w+a34*z
NOP          ;; make sure it does not
              ;; straddle across a cache line
FADDPST(2), ST;; a44*w+a34*z+a24*y
FADDPST(1), ST;; a44*w+a34*z+a24*y+a14*x
FSTPDWORD PTR [EDI+12];; store rw
ENDM

```

## ***B*** *Code Optimization*

---

# ***Appendix C***

## **AMD Processor Recognition**

### **Introduction**

---

Due to the increasing number of choices available in the x86 processor marketplace, the need for a simple way for hardware and software to identify the type of processor and its feature set has become critical. The CUID instruction was added to the x86 instruction set for this purpose.

The CUID instruction provides complete information about the processor (vendor, type, name, etc.) and its capabilities (features). After detecting the processor and its capabilities, software can be accurately tuned to the system for maximum performance and benefit to users. For example, game software can test the performance level available from a particular processor by detecting the type or speed of the processor. If the performance level is high enough, the software can enable additional capabilities or more advanced algorithms. Another example involves testing for the presence of MMX and 3D instructions on the processor. If the software finds this feature present when it checks the feature bits, it can utilize these more powerful extensions for dramatically better performance on new multimedia software.

# **C** *AMD Processor Recognition*

---

## **Using the CPUID Instruction**

---

### **Overview**

Software operating at any privilege level can execute the CPUID instruction to identify the processor and its feature set. In addition, the CPUID instruction implements multiple functions, each providing different information about the processor, including the vendor, model number, revision (stepping), features, cache organization, and processor name. The multiple-function approach allows the CPUID instruction to return a complete picture about the type of processor and its capabilities—more detailed information than could be returned by a single function. In addition to gathering all the information by calling multiple functions, the CPUID instruction provides the flexibility of making only one call to obtain the specific data requested once the processor vendor has been identified.

The functions are divided into two types: standard functions and extended functions. Standard functions provide a simple method for software to access information common to all x86 processors. Extended functions provide information on extensions specific to a vendor's processor (for example, AMD's processors).

The flexibility of the CPUID instruction allows for the addition of new CPUID functions in future generations of processors. See page 515 for a detailed description of the CPUID instruction.

### **Testing for the CPUID Instruction**

Beginning with the Am486®DX4 processor, all AMD processors implement the CPUID instruction. In order to avoid an invalid opcode exception on those processors that do not support the CPUID instruction, software must first test to determine if the CPUID instruction is present on the processor. The presence of the CPUID instruction is indicated by the ID bit (21) in the EFLAGS register. If this bit is writeable, the CPUID instruction is implemented on the processor.

Software uses the PUSHFD and POPFD instructions to write to the ID bit in the EFLAGS register. After reading the ID bit, a comparison determines if this operation changed the value of the ID bit. If the value changed, the CUID instruction is available for identifying the processor and its features. The following code sample demonstrates the way a program uses the PUSHFD and POPFD instructions to test the ID bit.

```
pushfd          ; Save EFLAGS to stack
pop    eax      ; Store EFLAGS in EAX
mov    ebx, eax ; Save in EBX for testing later
xor    eax, 00200000h ; Switch bit 21
push   eax      ; Copy "changed" value to stack
popfd      ; Save "changed" EAX to EFLAGS
pushfd     ; Push EFLAGS to top of stack
pop    eax      ; Store EFLAGS in EAX
cmp    eax, ebx ; See if bit 21 has changed
jz     NO_CUID   ; If no change, no CUID
```

## Using CUID Functions

When software uses the CUID instruction to identify a processor, it is important that it uses the instruction appropriately. The instruction has been defined to make it easy to identify the type and features of x86 processors manufactured by many different vendors.

The standard functions (EAX=0 and EAX=1) are the same for all processors. Having standard functions simplifies software's task of testing for and implementing features common to x86 processors. Software can test for these features and, as new x86 processors are released, benefit from these capabilities immediately.

Extended functions are specific to a vendor's processor. These functions provide additional information about AMD processors that software can use to identify enhanced features and functions. To test for extended functions, software checks for "AuthenticAMD" in the vendor identification string returned by function 0 and for a non-zero value in the EAX register returned by function 8000\_0000h.

Within AMD's family of processors, different members can execute a different number of functions. Table 92 on page 508 summarizes the CUID functions currently implemented on AMD processors.



# C AMD Processor Recognition

**Table 92. Summary of CPUID Functions in AMD Processors**  
(See page 515 for detailed descriptions of the functions.)

Standard Function	Extended Function	Description	AMD-K5 Processor (model 0)	AMD-K5 Processor (model 1, 2, and 3)	AMD-K6 Processor (model 6, 7, and 8)	AMD-K6 Processor (model 9)
0	—	Vendor String and Largest Standard Function Value	X	X	X	X
1	—	Processor Signature and Standard Feature Bits	X	X	X	X
—	8000_0000h	Largest Extended Function Value	—	X	X	X
—	8000_0001h	Extended Processor Signature and Extended Feature Bits	—	X	X	X
—	8000_0002h	Processor Name	—	X	X	X
—	8000_0003h	Processor Name	—	X	X	X
—	8000_0004h	Processor Name	—	X	X	X
—	8000_0005h	L1 Cache Information	—	X	X	X
—	8000_0006h	L2 Cache Information	—	—	—	X
<b>Note:</b> Future versions of these processors may implement additional functions.						

## Identifying the Processor's Vendor

Software must execute the standard function EAX=0. The CPUID instruction returns a 12-character string that identifies the processor's vendor. The instruction also returns the largest standard function input value defined for the CPUID instruction on the processor.

For AMD processors, function 0 returns a vendor string of "AuthenticAMD". This string informs the software to follow AMD's definition for subsequent CPUID functions and the registers returned for those functions.

Once the software identifies the processor's vendor, it knows the definition for all the functions supplied by the CPUID instruction. By using these functions, the software obtains the processor information needed to properly tune its functionality to the capabilities of the processor.

## Determining the Processor Signature (Standard Function)

Standard function 1 (EAX=1) of the CUID instruction returns the standard processor signature and feature bits. The standard processor signature is returned in the EAX register and provides information regarding the specific revision (stepping) and model of the processor and the instruction family level supported by the processor. The revision level is used to determine if the processor requires the implementation of software workarounds. Figure 119 shows the contents of the EAX register obtained by function 1. Table 93 on page 510 summarizes the specific processor signature values returned for AMD processors.

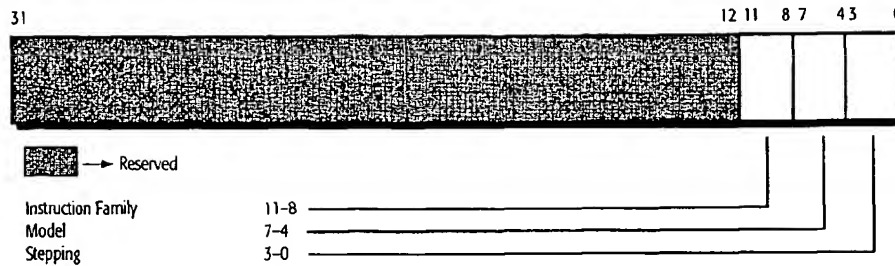


Figure 119. Contents of EAX Register Returned by Function 1

# C AMD Processor Recognition

**Table 93. Summary of Processor Signatures for AMD Processors**  
(See page 515 for details on bit locations and values.)

Processor	Instruction Family	Model	Stepping ID
Am486 and Am5x86 Processors	0100b (4h)	yyy <sup>2</sup>	xxx <sup>1</sup>
AMD-K5 Processor (Model 0)	0101b (5h)	0000b (0h)	xxx <sup>1</sup>
AMD-K5 Processor (Model 1)	0101b (5h)	0001b (1h)	xxx <sup>1</sup>
AMD-K5 Processor (Model 2)	0101b (5h)	0010b (2h)	xxx <sup>1</sup>
AMD-K5 Processor (Model 3)	0101b (5h)	0011b (3h)	xxx <sup>1</sup>
AMD-K6 Processor (Model 6)	0101b (5h)	0110b (6h)	xxx <sup>1</sup>
AMD-K6 Processor (Model 7)	0101b (5h)	0111b (7h)	xxx <sup>1</sup>
AMD-K6 3D Processor (Model 8)	0101b (5h)	1000b (8h)	xxx <sup>1</sup>
AMD-K6 3D+ Processor (Model 9)	0101b (5h)	1001b (9h)	xxx <sup>1</sup>
<b>Notes:</b> 1. Contact your AMD representative for the latest stepping information. 2. Model identifier information is provided in the AMD BIOS Development Guide, document number 19720.			

## Identifying Supported Features

The feature bits are returned in the EDX register for two CPUID functions—standard function 1 and extended function 8000\_0001h. Each bit corresponds to a specific feature and indicates if that feature is present on the processor. Table 94 on page 511 summarizes the standard feature bits, and Table 95 on page 512 summarizes the extended feature bits.

Before using any of the enhanced features added to the latest generation of processors, software should test each feature bit returned by functions 1 and 8000\_0001h to identify the capabilities available on the processor. For example, software must test bit 23 to determine if the processor executes MMX instructions. Attempting to execute an unavailable feature can cause errors and exceptions.

Bit 31, as returned by extended function 8000\_0001h, designates the presence of 3D technology. Other processor

vendors have adopted this technology so now bit 31 is considered an open standard. An alternate way to test for the presence of 3D technology (as opposed to testing for AuthenticAMD) is for software to implement the following algorithm:

1. Test for the CPUID instruction. (See “Testing for the CPUID Instruction” on page 506.)
2. Execute the CPUID extended function 8000\_0000h.
3. Test if the value returned in the EAX register is greater than or equal to 8000\_0000h.
4. Execute the CPUID extended function 8000\_0001h.
5. Test bit 31 in the EDX register for 3D technology.

**Table 94. Summary of Standard Feature Bits for AMD Processors**  
(See page 515 for details on bit locations and values.)

Feature	Description
Floating-Point Unit	A floating-point unit is available.
Virtual Mode Extensions	Virtual mode extensions are available.
Debugging Extensions	I/O breakpoint debug extensions are supported.
Page Size Extensions	4-Mbyte pages are supported.
Time Stamp Counter (with RDTSC and CR4 disable bit)	A time stamp counter is available in the processor, and the RDTSC instruction is supported.
K86 Model-Specific Registers (with RDMSR and WRMSR)	The K86 model-specific registers are available in the processor, and the RDMSR and WRMSR instructions are supported.
Machine Check Exception	The machine check exception is supported.
CMPXCHG8B Instruction	The CMPXCHG8B instruction is supported.
APIC	A local APIC unit is available.
Global Paging Extension	Global paging extensions are available.
Conditional Move Instructions	The conditional move instructions CMOV, FCMOV, and FCOMI are supported.
MMX Instructions	MMX instructions are supported.

# C AMD Processor Recognition

**Table 95. Summary of Extended Feature Bits for AMD Processors**  
(See page 515 for details on bit locations and values.)

Feature	Description
Floating-Point Unit	A floating-point unit is available.
Virtual Mode Extensions	Virtual mode extensions are available.
Debugging Extensions	I/O breakpoint debug extensions are supported.
Page Size Extensions	4-Mbyte pages are supported.
Time Stamp Counter (with RDTSC and CR4 disable bit)	A time stamp counter is available in the processor, and the RDTSC instruction is supported.
K86 Model-Specific Registers (with RDMSR and WRMSR)	The K86 model-specific registers are available in the processor, and the RDMSR and WRMSR instructions are supported.
Machine Check Exception	The machine check exception is supported.
CMPXCHG8B Instruction	The CMPXCHG8B instruction is supported.
Global Paging Extension	Global paging extensions are available.
SYSCALL and SYSRET Instructions	The SYSCALL and SYSRET instructions and associated extensions are supported.
Integer Conditional Move Instruction	The integer conditional move instruction CMOV is supported.
Floating-Point Conditional Move Instructions	The floating-point conditional move instructions FCMOV and FCOMI are supported.
MMX Instructions	MMX instructions are supported.
3D Instructions	3D instructions are supported.

## Testing For Extended Functions

Once software has identified the processor's vendor as AMD, it must test for extended functions by executing function 8000\_0000h. The EAX register returns the largest extended function input value defined for the CUID instruction on the processor. If this value is non-zero, extended functions are supported.

To simplify identifying processors and their features, the AMD extended functions include all the information provided in the standard functions as well as the additional AMD-specific feature enhancements. This duplication can minimize the number of function calls required by software. For more information, see 3D Feature Detection on page 83 and MMX Feature Detection on page 355

## Determining the Processor Signature (Extended Function)

Extended function 8000\_0001h returns the AMD processor signature. The signature is returned in the EAX register and provides generation, model, and stepping information for AMD processors. Figure 120 shows the contents returned in the EAX register.

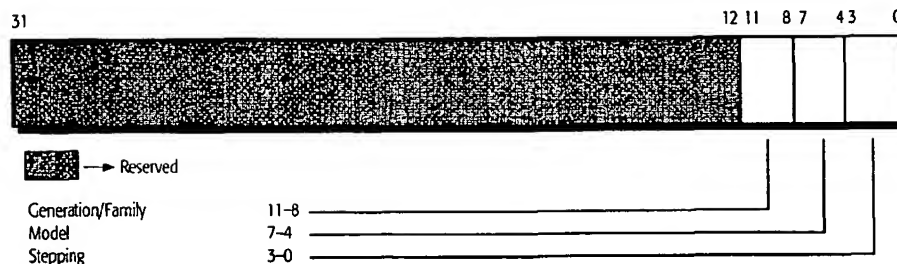


Figure 120. Contents of EAX Register Returned by Extended Function 8000\_0001h

# **C** *AMD Processor Recognition*

---

## **Displaying the Processor's Name**

Functions 8000\_0002h, 8000\_0003h, and 8000\_0004h return an ASCII string containing the name of the processor. These functions eliminate the need for software to search for the processor name in a lookup table, a process requiring a large block of memory and frequent updates. Instead, software can simply call these three functions to obtain the name string (48 ASCII characters in little endian format) and display it on the screen. Although the name string can be up to 48 characters in length, shorter names have the remaining byte locations filled with the ASCII NULL character (00h). To simplify the display routines and avoid using screen space, software only needs to display characters until a NULL character is detected.

## **Displaying Cache Information**

Functions 8000\_0005h and 8000\_0006h (function 8000\_0006h is only supported in AMD-K6<sup>®</sup> 3D+ Model 9) provide cache information for the processor. Some diagnostic software displays information about the system and the processor's configuration. It is common for this type of software to provide cache size and organization of information. Functions 8000\_0005h and 8000\_0006h provide a simple way for software to obtain information about the on-chip cache and TLB structures. The size and organization information is returned in the registers as described on page 515. Software can simply display these values, eliminating the need for large pieces of code to test the memory structures.

## **Sample Code**

---

A code sample that uses the CUID instruction to identify the processor and its features is available from AMD's website at <http://www.amd.com/k6/k6docs/>.



## CPUID

<i>mnemonic</i>	<i>opcode</i>	<i>description</i>
CPUID	0F A2h	Identify the processor and its feature set
Privilege:	none	
Registers Affected:	EAX, EBX, ECX, EDX	
Flags Affected:	none	
Exceptions Generated:	none	

The CPUID instruction is an application-level instruction that software executes to identify the processor and its feature set. This instruction offers multiple functions, each providing a different set of information about the processor. The CPUID instruction can be executed from any privilege level. Software can use the information returned by this instruction to tune its functionality for the specific processor and its features.

Not all processors implement the CPUID instruction. Therefore, software must test to determine if the instruction is present on the processor. If the ID bit (21) in the EFLAGS register is writeable, the CPUID instruction is implemented.

The CPUID instruction supports multiple functions. The information associated with each function is obtained by executing the CPUID instruction with the function number in the EAX register. Functions are divided into two types: standard functions and extended functions. Standard functions are found in the low function space, 0000\_0000h–7FFF\_FFFFh. In general, all x86 processors have the same standard function definitions.

Extended functions are defined specifically for processors supplied by the vendor listed in the vendor identification string. Extended functions are found in the high function space, 8000\_0000h–8FFF\_FFFFh. Because not all vendors have defined extended functions, software must test for their presence on the processor.

AMD processors have extended functions under the following conditions:

- The processor returns the “AuthenticAMD” vendor identification string.
- The 8000\_0000h function returns a non-zero value in the EAX register.



# C AMD Processor Recognition

## Standard Functions

### Function 0 – Largest Standard Function Input Value and Vendor Identification String

*Input:* EAX = 0

*Output:* EAX = Largest function input value recognized by the CPUID instruction  
EBX, EDX, ECX = Vendor identification string

This is a standard function found in all processors implementing the CPUID instruction. It returns two values. The first value is returned in the EAX register and indicates the largest standard function value recognized by the processor. The second value is the vendor identification string. This 12-character ASCII string is returned in the EBX, EDX, and ECX registers in little endian format.

AMD processors return a vendor identification string of “AuthenticAMD” that software uses as follows:

- To identify the processor as an AMD processor
- To apply AMD’s definition of the CPUID instruction for all additional function calls

### Function 1 – Processor Signature and Standard Feature Flags

*Input:* EAX = 1

*Output:* EAX = Processor Signature  
EBX = Reserved  
ECX = Reserved  
EDX = Standard Feature Flags

Function 1 returns two values—the Processor Signature and the Standard Feature Flags. The processor signature is returned in the EAX register and identifies the specific processor by providing information on its type—instruction family, model, and revision (stepping). The information is formatted as follows:

- EAX[3–0] Stepping ID
- EAX[7–4] Model
- EAX[11–8] Instruction Family
- EAX[31–12] Reserved

The standard feature flags are returned in the EDX register and indicate the presence of specific features. In most cases, a “1” indicates the feature is present, and a “0” indicates the feature is not present. Table 96 on page 517 contains a list of the currently defined standard feature flags. Reserved bits will be used for new features as they are added.

**Table 96. Standard Feature Flag Descriptions**

Bit	Feature	Description
0	Floating-Point Unit	0 = No FPU 1 = FPU Present
1	Virtual Mode Extensions	0 = No Support 1 = Support
2	Debugging Extensions	0 = No Support 1 = Support
3	Page Size Extensions	0 = No Support 1 = Support 4Mbyte Pages
4	Time Stamp Counter (with RDTSC and CR4 disable bit)	0 = No Support 1 = Support
5	K86 Model-Specific Registers (with RDMSR and WRMSR)	0 = No Support 1 = Support
6	Reserved	—
7	Machine Check Exception	0 = No Support 1 = Support
8	CMPXCHG8B Instruction	0 = No Support 1 = Support
9	APIC*	0 = No Support 1 = Support
10–11	Reserved	—
12	Memory Type Range Registers	0 = No Support 1 = Support
13	Global Paging Extension*	0 = No Support 1 = Support
14	Reserved	—
15	Conditional Move Instruction	0 = No Support 1 = Support
16–22	Reserved	—
23	MMX Instructions	0 = No Support 1 = Support
24–31	Reserved	—

**Note:**  
\* The AMD-K5 processor (model 0) reserves bit 13 and implements feature bit 9 to indicate support for Global Paging Extensions instead of support for APIC.

# C AMD Processor Recognition

## Extended Functions

### Function 8000\_0000h – Largest Extended Function Input Value

**Input:** EAX = 8000\_0000h

**Output:** EAX = Largest function input value recognized by the CPUID instruction  
EBX = Reserved  
ECX = Reserved  
EDX = Reserved

Function 8000\_0000h returns a value in the EAX register that indicates the largest extended function value recognized by the processor.

### Function 8000\_0001h – AMD Processor Signature and Extended Feature Flags

**Input:** EAX = 8000\_0001h

**Output:** EAX = AMD Processor Signature  
EBX = Reserved  
ECX = Reserved  
EDX = Extended Feature Flags

Function 8000\_0001h returns two values—the AMD Processor Signature and the Extended Feature Flags. The AMD processor signature is returned in the EAX register and identifies the specific processor by providing information regarding its type—generation/family, model, and revision (stepping). The information is formatted as follows:

- EAX[3–0] Stepping ID
- EAX[7–4] Model
- EAX[11–8] Generation/Family
- EAX[31–12] Reserved

The extended feature flags are returned in the EDX register and indicate the presence of specific features found in AMD processors. In most cases, a “1” indicates the feature is present, and a “0” indicates the feature is not present. Table 97 on page 519 contains a list of the currently defined extended feature flags. Reserved bits will be used for new features as they are added.

**Table 97. Extended Feature Flag Descriptions**

Bit	Feature	Description
0	Floating-Point Unit	0 = No FPU 1 = FPU Present
1	Virtual Mode Extensions	0 = No Support 1 = Support
2	Debugging Extensions	0 = No Support 1 = Support
3	Page Size Extensions	0 = No Support 1 = Support 4Mbyte Pages
4	Time Stamp Counter (with RDTSC and CR4 disable bit)	0 = No Support 1 = Support
5	K86 Model-Specific Registers (with RDMSR and WRMSR)	0 = No Support 1 = Support
6	Reserved	—
7	Machine Check Exception	0 = No Support 1 = Support
8	CMPXCHG8B Instruction	0 = No Support 1 = Support
9–10	Reserved	—
11	SYSCALL and SYSRET Instructions	0 = No Support 1 = Support
12	Reserved	—
13	Global Paging Extension	0 = No Support 1 = Support
14	Reserved	—
15	Integer Conditional Move Instruction	0 = No Support 1 = Support
16	Floating-Point Conditional Move Instructions	0 = No Support 1 = Support
17–22	Reserved	—
23	MMX Instructions	0 = No Support 1 = Support
24–30	Reserved	—
31	3D Instructions	0 = No Support 1 = Support

# C *AMD Processor Recognition*

## **Functions 8000\_0002h, 8000\_0003h, and 8000\_0004h – Processor Name String**

*Input:* EAX = 8000\_0002h, 8000\_0003h, or 8000\_0004h

*Output:* EAX = Processor Name String  
EBX = Processor Name String  
ECX = Processor Name String  
EDX = Processor Name String

Functions 8000\_0002h, 8000\_0003h, and 8000\_0004h each return part of the processor name string in the EAX, EBX, ECX, and EDX registers. These three functions use the four registers to return an ASCII string of up to 48 characters in little endian format. For example, function 8000\_0002h returns the first 16 characters of the processor name. The first character resides in the least significant byte of EAX, and the last character (of this group of 16) resides in the most significant byte of EDX. The NULL character (ASCII 00h) is used to indicate the end of the processor name string. This feature is useful for processor names that require fewer than 48 characters.

## **Function 8000\_0005h – L1 Cache Information**

*Input:* EAX = 8000\_0005h

*Output:* EAX = Reserved  
EBX = TLB Information  
ECX = L1 Data Cache Information  
EDX = L1 Instruction Cache Information

Function 8000\_0005h returns information about the processor's on-chip L1 caches and associated TLBs. Tables 98, 99, and 100 on page 521 provide the format for the information returned by the 8000\_0005h function.

**Table 98. EBX Format Returned by Function 8000\_0005h**

	Data TLB		Instruction TLB	
	Associativity*	# Entries	Associativity*	# Entries
EBX	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
<b>Note:</b> * Full associativity is indicated by a value of 0FFh.				

**Table 99. ECX Format Returned by Function 8000\_0005h**

	L1 Data Cache			
	Size (Kbytes)	Associativity*	Lines per Tag	Line Size (bytes)
ECX	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
<b>Note:</b> * Full associativity is indicated by a value of 0FFh.				

**Table 100. EDX Format Returned by Function 8000\_0005h**

	L1 Instruction Cache			
	Size (Kbytes)	Associativity*	Lines per Tag	Line Size (bytes)
EDX	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0
<b>Note:</b> * Full associativity is indicated by a value of 0FFh.				

# C AMD Processor Recognition

## Function 8000\_0006h – L2 Cache Information

This function is available on the AMD-K6 3D+ processor Model 9.

**Input:** EAX = 8000\_0006h

**Output:** EAX = Reserved  
EBX = Reserved  
ECX = L2 Unified Cache Information  
EDX = Reserved

Function 8000\_0006h returns information about the processor's L2 cache. Table 10 provides the format for the information returned by the 8000\_0006h function.

**Table 101. ECX Format Returned by Function 8000\_0006h**

	L2 Cache			
	Size (Kbytes)	Associativity*	Lines per Tag	Line Size (bytes)
ECX	Bits 31–16	Bits 15–12	Bits 11–8	Bits 7–0
<b>Note:</b> * Full associativity is indicated by a value of 0Fh.				

## Values Returned by the CPUID Instruction

Table 102 contains the values returned by the CPUID instruction for AMD processors models 6 through 9.

**Table 102. Values Returned By AMD Processors**

Function Register	AMD-K6 Processor (Model 6)	AMD-K6 Processor (Model 7)	AMD-K6 3D Processor (Model 8)	AMD-K6 3D+ Processor (Model 9)
Function: 0				
EAX	0000_0001h	0000_0001h	0000_0001h	0000_0001h
EBX	6874_7541h	6874_7541h	6874_7541h	6874_7541h
ECX	444D_4163h	444D_4163h	444D_4163h	444D_4163h
EDX	6974_6E65h	6974_6E65h	6974_6E65h	6974_6E65h
Function: 1				
EAX	0000_056Xh	0000_057Xh	0000_058Xh	0000_059Xh
EBX	Reserved	Reserved	Reserved	Reserved
ECX	Reserved	Reserved	Reserved	Reserved
EDX	0080_01BFh	0080_01BFh	0080_01BFh	0080_21BFh
Function: 8000_0000h				
EAX	8000_0005h	8000_0005h	8000_0005h	8000_0006h
EBX	Reserved	Reserved	Reserved	Reserved
ECX	Reserved	Reserved	Reserved	Reserved
EDX	Reserved	Reserved	Reserved	Reserved
Function: 8000_0001h				
EAX	0000_066Xh	0000_067Xh	0000_068Xh	0000_069Xh
EBX	Reserved	Reserved	Reserved	Reserved
ECX	Reserved	Reserved	Reserved	Reserved
EDX	0080_01BFh	0080_09BFh*	8080_09BFh	8080_29BFh
Function: 8000_0002h				
EAX	2D44_4D41h	2D44_4D41h	2D44_4D41h	2D44_4D41h
EBX	6D74_364Bh	6D74_364Bh	7428_364Bh	7428_364Bh
ECX	202F_7720h	202F_7720h	3320_296Dh	3320_296Dh
EDX	746C_756Dh	746C_756Dh	7270_2044h	5020_2B44h
<b>Note:</b> * A value of 0080_01BFh is returned by the AMD-K6 processor Model 7 with an A stepping (a 0000_0570h value is returned in EAX by CPUID Function 1).				



# C AMD Processor Recognition

**Table 102. Values Returned By AMD Processors (continued)**

Function Register	AMD-K6 Processor (Model 6)	AMD-K6 Processor (Model 7)	AMD-K6 3D Processor (Model 8)	AMD-K6 3D+ Processor (Model 9)
Function: 8000_0003h				
EAX	6465_6D69h	6465_6D69h	7365_636Fh	6563_6F72h
EBX	6520_6169h	6520_6169h	0072_6F73h	726F_7373h
ECX	6E65_7478h	6E65_7478h	0000_0000h	0000_0000h
EDX	6E6F_6973h	6E6F_6973h	0000_0000h	0000_0000h
Function: 8000_0004h				
EAX	0000_0073h	0000_0073h	0000_0000h	0000_0000h
EBX	0000_0000h	0000_0000h	0000_0000h	0000_0000h
ECX	0000_0000h	0000_0000h	0000_0000h	0000_0000h
EDX	0000_0000h	0000_0000h	0000_0000h	0000_0000h
Function: 8000_0005h				
EAX	Reserved	Reserved	Reserved	Reserved
EBX	0280_0140h	0280_0140h	0280_0140h	0280_0140h
ECX	2002_0220h	2002_0220h	2002_0220h	2002_0220h
EDX	2002_0220h	2002_0220h	2002_0220h	2002_0220h
Function: 8000_0006h				
EAX	Undefined	Undefined	Undefined	Reserved
EBX	Undefined	Undefined	Undefined	Reserved
ECX	Undefined	Undefined	Undefined	0100_4220h
EDX	Undefined	Undefined	Undefined	Reserved
<b>Note:</b> * A value of 0080_01BFh is returned by the AMD-K6 processor Model 7 with an A stepping (a 0000_0570h value is returned in EAX by CPUID Function 1).				

# Index

## Numerics

100-MHz Bus	xxv, 1, 3
clock switching characteristics	314
input setup and hold timings	318
output delay timings	316
321-Pin Staggered CPGA Package	1
specification	343–344
3D Technology	xxv, 1, 8, 15, 18–20, 23, 52
	79, 81–135, 174, 227, 482, 505
data types	32, 85
definitions	88
exceptions	93, 256
execution resources	89
execution unit	18, 255
exponent ranges	89
feature detection	83, 512
instruction coding	95
instruction compatibility, floating-point and	84, 256
instruction formats	87
instructions	79, 87, 95–135
key functionality	82
matrix multiplication optimization example	487
prefixes	94
registers	31, 84, 256
task switching	93
technology	9
66-MHz Bus	1
clock switching characteristics	315
input setup and hold timings	322
output delay timings	320

## A

A[20:3]	310
A[31:3]	141
A20M#	140, 258
A20M# Masking of Cache Accesses	251
Absolute Ratings	304
Accelerated Graphic Port (AGP)	1, 3
Acknowledge, Interrupt	218
Address	
bus	141–146, 156, 188, 207, 210, 212, 247
hold	143
parity	144
parity check	145
stack, return	22
ADS#	142, 310
ADSC#	142
AGP	xxv, 1, 3
AHOLD	143, 292
-initiated inquire hit to modified line	210
-initiated inquire hit to shared or exclusive line	209
-initiated inquire miss	207
restriction	212
Airflow	
consideration, layout and	335
management	336

Allocate, Write	240
AMD Processor	
recognition	505
signature	513, 518
AMD-K6	
family of processors	456
processor x86 coding optimizations, general	474
AMD-K7	xxvi
AP	144
APCHK#	145
APIC	511, 517
Application Note	
I/O model	312
SYSCALL and SYSRET instructions	41
write allocate	243
Architecture	2
cache	233
internal	5–22, 455
MMX multimedia technology	348
x86	23, 48, 81, 359, 455
Asserted	139

## B

Backoff	148
Base Address, SMM	263
BE[7:0]#	146
BF[2:0]	147, 228, 296
BIST	270
Bits, Predecode	11, 235
Block Diagram	6–7
BOFF#	148, 213
locked operation with	216
Boundary Scan	
register (BSR)	273
test access port (TAP)	271
BR	277
Branch	
condition unit	464
execution unit	22
history table	21
logic	9
prediction	1–2, 9, 22
prediction logic	20–21, 456, 464, 473, 477
target cache	21
BRDY#	149
BRDYC#	150, 228, 310
BREQ	151
BSR	273
Buffer	
characteristics, I/O	309
model, I/O	310
Built-In Self-Test (BIST)	270
Burst	
reads	195
ready	149
ready copy	150, 228
writeback	197

<b>Bus</b>	
100-MHz	xxv, 1, 3, 314, 316, 318
66-MHz	1, 315, 320, 322
address	143–146, 156, 188, 207, 210, 212, 247
arbitration cycles, inquire and	201
backoff	213
cycles	187
cycles, special	220
data	143, 146, 149, 154–155, 172, 175
	190–192, 207, 212, 215
enables	146
frequency	147
hold request	162
lock	167
request	151
state machine diagram	189
<b>Bus States</b>	
address	191
data	190
data-NA# requested	190
idle	190
pipeline address	191
pipeline data	191
transition	192
<b>BYPASS Instruction</b>	278
<b>Bypass Register</b>	277
<b>C</b>	
<b>Cache</b>	10, 82, 92, 134–135, 462, 465, 473, 476
	482–483, 491, 495–496, 500, 502, 506
	514, 520–521
branch target	21
coherency	247
disabling	238
enable	166
flush	159
inhibit, L1	282
L1	1, 40, 92, 134, 198, 233, 240, 244, 247
	251, 269–270, 282, 473, 491, 493, 520–521
L2	xxv, 282, 496, 522
MESI states in the data	235
operation	235
organization	233
snooping	250
states	246
writeback	6, 10
<b>CACHE#</b>	152, 237
<b>Cacheable</b>	
access	152
page, write to a	241
<b>Cache-Line</b>	
fills	239
replacement	240, 248
<b>Cache-Related Signals</b>	238
<b>Capture-DR state</b>	280
<b>Capture-IR state</b>	280
<b>Case Temperature</b>	334
<b>CD-ROM</b>	xxv, xxviii
<b>Centralized Scheduler</b>	16
<b>Ceramic Pin Grid Array (CPGA)</b>	1–2, 343, 346

<b>Characteristics</b>	
I/O buffer	309
I/O buffer AC and DC	312
<b>CLK</b>	152
<b>Clock</b>	82, 152
control	291
<b>Clock States</b>	
halt	292
stop clock	223, 295–296
stop grant	223, 293
stop grant inquire	295
<b>Code Optimization</b>	90, 455–503
<b>Code Sample</b>	
3D technology	487
analysis	466
CPUID instruction	83, 355, 507, 514
floating-point	502
non-optimized	488
optimization	486–487
optimized	490
<b>Coding Guidelines, Optimization</b>	472
<b>Coding Optimizations</b>	
general AMD-K6 3D processor x86	474
integer x86	478
multimedia	482
the AMD-K6 3D processor	456
the AMD-K6 family of processors	456
<b>Coherency States, Writethrough vs. Writeback</b>	251
<b>Coherency, Cache</b>	247
<b>Compatibility, Floating-Point, MMX, and</b>	
3D Instructions	84, 256, 457
<b>Configuration and Initialization, Power-on</b>	227
<b>Connection Requirements, Pin</b>	301
<b>Connections, Power</b>	299
<b>Constraints, Resource</b>	466
<b>Control</b>	
register	34, 93
unit, scheduler/instruction	8
<b>Cooperative Multitasking</b>	356
<b>Counter, Time Stamp</b>	40, 511–512, 517, 519
<b>CPGA</b>	1–2, 343, 346
<b>CPUID Instruction</b>	83, 366, 369, 505–511
	514–516, 518, 523
<b>Cycle</b>	
hold and hold acknowledge	202
shutdown	222
<b>Cycles</b>	
bus	187
inquire	140–145, 156, 160–161, 177, 183, 198
	201, 203, 205, 207, 209–210, 212–213, 216
	247–251, 282, 291–293, 295
inquire and bus arbitration	201
interrupt acknowledge	141, 144, 146, 153, 170, 182
locked	215
pipelined	10, 142
pipelined write	154
special bus	220
writeback	140, 142–143, 157, 160, 183, 198
	205, 209–210, 212, 214, 216, 236–237, 282, 295

## D

D/C#	153
D[63:0]	154
Data	
bus	143, 146, 149, 154–155, 172, 175
	190–192, 207, 212, 215
cache, MESI states in the	92, 235
parity	155
Data Types	86
3D	32, 85
floating-point register	30
integer	25
MMX	31, 352
Data/Code	153
DC Characteristics	305
Debug	283, 511
exceptions	288
Debug Registers	36, 283
DR3–DR0	286
DR5–DR4	286
DR6	287
DR7	287
Decode, Instruction	13, 90, 456
Decoders	7
Decoupling Recommendations	301
Dependency Latencies, Execution Units and	458
Descriptions, Signal	137
Design, Thermal	331
Designations, Pin	342
Device Identification Register	276
Diagram, Pin Description	340
Diagrams, Timing	187–226
DIR	276
Disabling, Cache	238
Displaying	
cache information	514
the processor's name	514
Dissipation, Power	307
Divide	
optimized 15-bit precision	498
optimized full 24-bit precision	498
Divides, Pipelined Pair of 24-Bit Precision	498
Division	95, 497
square root	95, 498
DP[7:0]	155
DR3–DR0	286
DR5–DR4	286
DR6	287
DR7	287
Drive Strength, Selectable	310
Driven	139

## E

EADS#	156
EFER	39, 41, 231
EFLAGS Register	33, 83, 506–507
Electrical Data	303
EMMS Instruction	85, 359, 361, 474, 482
Environment, Software	23
EWBE#	157, 292

Exception	144–145, 155, 158, 172
	222, 256, 267, 287–289, 358, 475, 511
flags	28–29
floating-point	158, 163, 254, 256
handler	283
machine check	39, 511–512, 517, 519
Exceptions	
3D	89, 93–94, 256
and interrupts	51
debug	288
floating-point	254
handling floating-point	254
interrupts, and debug in SMM	267
MMX	256, 358
Execution Resources, 3D	89
Execution Unit	16, 82
3D	18, 84, 255, 460, 482, 500
branch	22
floating-point	2, 27, 82, 253, 464, 482
	500, 502, 511–512, 517, 519
integer X	18
integer Y	18
load	91, 462, 467, 469–471, 489, 491–492
multimedia	2, 18, 84, 255, 460
store	463, 467, 470, 490, 493
terminology	458
see also Unit	
Execution Units	1, 7, 18, 456, 465–466
and Dependency Latencies	458
register	460
Extended Functions	83, 506–507, 513, 515, 518
External	
address strobe	156
write buffer empty	157
EXTTEST Instruction	277

## F

Feature Detection	510
3D	83
MMX	355
FEMMS Instruction	82, 85, 87, 91–92, 96, 474, 482
FERR#	158, 254, 256
Fetch Unit	12
Fetch, Instruction	12
Float Conditions	181, 185
Floated	139
Floating-Point	
and MMX/3D instruction compatibility	256
and multimedia execution units	253
code sample	502
error	158
handling exceptions	254
register data types	30
registers	27
unit	253, 464, 511–512, 517, 519
FLUSH#	159, 227, 248, 251, 270, 292
Frequency	296, 314–315, 326
multiplier	152
operating	147, 152, 228
FRSTOR	356
FSAVE	356
Function 0	83, 507–508, 516

Function 1	83, 509–510, 516
Function 8000_0000h	507, 511, 513, 518
largest standard function	518
Function 8000_0001h	510, 513, 518
processor signature	518
Function 8000_0005h	514, 520–521
cache information	520
Functions 8000_0002h, 8000_0003h, and 8000_0004h	514, 520
processor name string	520

## G

Gate Descriptor	48, 51
General-Purpose Registers	23
Grounding, Power and	299, 339

## H

Halt State	292
Handling Floating-Point Exceptions	254
Heat Dissipation Path	334
HIGHZ Instruction	278
History Table, Branch	21
Hit to	
modified line	160
modified line, AHOLD-initiated inquire	210
modified line, HOLD-initiated inquire	205
shared or exclusive line, AHOLD-initiated inquire	209
shared or exclusive line, HOLD-initiated inquire	203
HIT#	160
HITM#	160, 310
HLDA	161
HOLD	162
-initiated inquire hit to modified line	205
-initiated inquire hit to shared or exclusive line	203
Hold	
acknowledge	161, 202–203
acknowledge cycle	202
timing	313, 328

## I

I/O	
buffer AC and DC characteristics	312
buffer characteristics	309
buffer model	310
misaligned read and write	200
model application note	312
read and write	199
trap dword	265
trap restart slot	266
IBIS	310
IDCODE instruction	278
Identifying	
supported features	510
the processor's vendor	508
IEEE	85, 88
IEEE 1149.1	1, 271
IEEE 754	1, 27, 253
IEEE 854	253
IGNNE#	163, 254, 256
Ignore Numeric Exception	163

INTT	164, 292
-initiated transition from protected mode to	
real mode	225
state of processor after	232
Initialization	164
power-on configuration and	227
Input Setup and Hold Timings	
for 100-MHz bus operation	318
for 66-MHz bus operation	322
Inquire	204, 206, 208, 291
bus arbitration cycles	201
cycle hit	160
cycles	140–145, 156, 160–161, 177, 183
198, 201, 203, 205, 207, 209–210	
212–213, 216, 247–251, 282, 291–293, 295	
miss, AHOLD-initiated	207
Instruction	
decode	13, 456
fetch	12
formats, 3D	87
formats, MMX	354
pointer	27
prefetch	10
Instructions	52–81
3D	79, 82–84, 87–135, 255, 512, 519
CPUID	83, 366, 369, 505–508, 510–511
514–516, 518, 523	
EMMS	15, 85, 359, 361, 474, 482
FEMMS	15, 82, 87, 96, 474, 482
FERR#	254, 256
FLUSH#	251
FRSTOR	356
FSAVE	356
IGNNE#	254, 256
INVD	248
MMX	75, 87, 255, 353, 360–453
MOVD	352, 362
MOVQ	352, 363
PACKSSDW	364
PACKUSWB	369
PADDB	372
PADDD	374
PADDSB	376
PADDSW	378
PADDUSB	380
PADDUSW	382
PADDW	384
PAND	386
PANDN	388
PAVGUSB	97
PCMPEQB	390
PCMPEQD	392
PCMPEQW	394
PCMPGTB	396
PCMPGTD	398
PCMPGTW	400
PF2ID	99
PFACC	101
PFADD	103
PFCMPEQ	105
PFCMPGE	107
PFCMPGT	109
PFMAX	111
PFMIN	113
PFMUL	87, 115

PFRCP	117
PFRCPIT1	119
PFRCPIT2	121
PFRSQIT1	123
PFRSQRT	125
PFSUB	127
PFSUBR	129
PI2FD	131
PMADDWD	349, 402
PMULHRW	132
PMULHW	404
PMULLW	406
POPDF	507, 512
POR	408
PREFETCH	11, 87, 134, 482, 512
PREFETCHW	134
PSLLD	410
PSLLQ	412
PSLLW	414
PSRAD	416
PSRAW	418
PSRLD	420
PSRLQ	422
PSRLW	424
PSUBB	426
PSUBD	428
PSUBSB	430
PSUBSW	432
PSUBUSB	434
PSUBUSW	436
PSUBW	438
PUNPCKHBW	440
PUNPCKHDQ	442
PUNPCKHWD	444
PUNPCKLBW	446
PUNPCKLDQ	448
PUNPCKLWD	450
PUSHFD	507
PXOR	452
supported by the AMD-K6 3D processor	52
SYSCALL	519
SYSRET	519
TAP	277
WBINVD	248, 251
Integer	
data types	25, 86
X execution unit	18
x86 coding optimizations	478
Y execution unit	18
Internal	
architecture	5–22
snooping	247
Interrupt	165, 176, 218, 222–223, 225
232, 254, 256, 258, 267, 288, 294	
acknowledge	141, 149, 153, 165, 167, 172, 215, 218
acknowledge cycles	141, 144, 146, 153, 170, 182
descriptor table register	42–43
flag	33, 165, 176
gate	50
redirection bitmap	44
request	165
service routine	165, 170, 254, 257
system management	257
type of	51

Interrupts	
01h	289
03h	289
10h	254
exceptions and	51
INTR	165
IRQ13	255
NMI	170
INTR	165, 292
INV	165
Invalidation Request	165
INVD Instruction	248

## K

KEN#	166
Key Functionality	
3D	82
MMX	348

## L

L1 Cache	1, 40, 92, 134, 198, 233
240, 244, 247, 251, 269–270, 282	
473, 491, 493, 520–521	
inhibit	282
L2 Cache	xxv, 282, 496, 522
Latencies	
and throughput	465
execution units and dependency	458
Level-One Cache. See L1 Cache	
Limit, Write Allocate	242
Line Fills, Cache	239
Load Unit	91, 462, 467, 469–471, 489, 491–492
LOCK#	167
Locked	
cycles	215
operation with BOFF# intervention	216
operation, basic	215
Logic	
branch	9
branch prediction	456, 464, 473, 477
branch-prediction	20–21, 456, 464, 473, 477
external support of floating-point exceptions	254

## M

M/IO#	168
Machine Check Exception	39, 511–512, 517, 519
Maskable Interrupt	165
Matrix Multiplication Optimization Example	487
MCAR	39, 231
MCTR	39–40, 231
Memory	
or I/O	168
read and write, misaligned single-transfer	194
read and write, single-transfer	192
reads and writes	192
MESI	1, 10, 201, 205, 234, 246, 249, 251
bit	11, 235–236, 495
states in the data cache	235
Microarchitecture	2, 82, 455–457
enhanced RISC86	6
overview, AMD-K6 3D processor	5

Misaligned	
I/O read and write	200
single-transfer memory read and write	194
Mixing MMX and Floating-Point Instructions	358
MMX	xxvi, 8, 15–16, 23, 52, 82, 84, 87, 89–91
	93–94, 174, 227, 505, 510–511, 517, 519
data types	31, 352
exceptions	256, 358
feature detection	355
instruction compatibility, floating-point and	256
instruction formats	354
instruction set	360
instructions	82, 84, 87, 89, 91–94, 353, 360–453
	505, 510–512, 517, 519
key functionality	348
multimedia technology	347–453
multimedia technology architecture	348
prefixes	359
programming considerations	355
register set	350
registers	31, 84, 350
Mode, Tri-State Test	270
Model-Specific Registers (MSR)	39
ModR/M	
address mode	483, 496
byte	52–53, 71, 75, 79, 134, 482, 497
instruction	483
instruction format	87
MOVD Instruction	352, 362
MOVQ Instruction	352, 363
MPEC Decoding	82
MSR	39
Multimedia	
coding optimizations	482
execution unit	18, 255
technology, MMX	347–453
Multiplication	
optimization example	487

## N

NA#	169
Negated	139
Next Address	169
NML	170, 292
No-Connect Pins	175, 301
Non-Maskable Interrupt	170
Non-Pipelined	193, 239

## O

Operands	87–89, 458
Operating Ranges	303
Operation, Cache	235
Optimization	
code	455
coding guidelines	472
example, 3D matrix multiplication	487
techniques, general x86	472
Optimizations	
general AMD-K6 processor x86 coding	474
integer x86 coding	478
multimedia coding	482

Optimized	
15-bit precision divide	498
15-bit precision square root	499
24-bit precision square root	499
full 24-bit precision divide	498
Organization, Cache	233
Output	
delay timings for 100-MHz bus operation	316
delay timings for 66-MHz bus operation	320
signals	229

## P

Package	
specifications	343
thermal specifications	331
PACKSSDW Instruction	364
PACKSSWB Instruction	366
PACKUSWB Instruction	369
PADDB Instruction	372
PADD Instruction	374
PADDSB Instruction	376
PADDSD Instruction	378
PADDUSB Instruction	380
PADDUSW Instruction	382
PADDW Instruction	384
Page	
cache disable	171
directory entry (PDE)	47, 236
table entry (PTE)	47–48, 236
writethrough	173
Paging	45, 511–512, 517, 519
PAND Instruction	386
PANDN Instruction	388
Parity	138, 144, 146, 155, 172, 192
bit	144, 155, 172
check	144–145, 155, 172
error	145, 172, 207, 273
flags	33
PAVGUSB Instruction	92, 97
PCD	171, 236, 244
PCHK#	172
PCMPEQB Instruction	390
PCMPEQD Instruction	392
PCMPEQW Instruction	394
PCMPGTB Instruction	396
PCMPGTD Instruction	398
PCMPGTW Instruction	400
PF2ID Instruction	88, 90, 92, 99
PFACC Instruction	90, 92, 101
PFADD Instruction	90, 92, 103
PFCMPEQ Instruction	92, 105
PFCMPGE Instruction	92, 107
PFCMPGT Instruction	92, 109
PFMAX Instruction	90, 92, 111
PFCMPGT Instruction	90, 92, 113
PFMUL Instruction	87, 90, 92, 115
PFRCPI Instruction	90, 92, 117
PFRCPI1 Instruction	90, 92, 119
PFRCPI2 Instruction	90, 92, 121
PFRSQIT1 Instruction	90, 92, 123
PFRSQRT Instruction	90, 92, 125
PFSUB Instruction	90, 92, 127

PFSUBR Instruction	90, 92, 129
PI2FD Instruction	88, 90, 92, 131
Pin	
connection requirements	301
description diagram	340
designations	342
Pipeline	21, 82, 91, 191, 196
	457–460, 462–463, 466
control	20
register X and Y	20
six-stage	6, 8, 459
Pipelined	10, 19, 89, 169, 191, 195, 197
	212, 233, 245, 468, 471, 482, 498, 500, 502
burst reads	195
cycles	10, 142, 154
design	18
pair of 24-bit precision divides	498
PMADDWD Instruction	349, 402
PMULHRW Instruction	92, 132
PMULHW Instruction	404
PMULLW Instruction	406
Pointer, Instruction	27
POPFD Instruction	507
POR Instruction	408
Power	
and grounding	299, 339
connections	299
dissipation	307
Power-on Configuration and Initialization	227
Precision Divide, Optimized Full 24-Bit	498
Precision Divides, Pipelined Pair of 24-Bit	498
Precision Square Root	
optimized 15-bit	499
optimized 24-bit	499
Predecode Bits	10–11, 235
Prediction Logic, Branch	456, 464, 473, 477
Preemptive Multitasking	357
PREFETCH Instruction	11, 87, 134, 482
PREFETCH/PREFETCHW Instructions	134
Prefetching	10, 82, 245
PREFETCHW Instruction	134
Prefixes	
3D	94
MMX	359
Processors, The AMD-K6 Family of	456
Programming	
considerations, MMX	355
steps	493
PSILD Instruction	410
PSLLQ Instruction	412
PSLLW Instruction	414
PSRAD Instruction	416
PSRAW Instruction	418
PSRLD Instruction	420
PSRLQ Instruction	422
PSRLW Instruction	424
PSUBB Instruction	426
PSUBD Instruction	428
PSUBSB Instruction	430
PSUBSW Instruction	432
PSUBUSB Instruction	434
PSUBUSW Instruction	436
PSUBW Instruction	438
PUNPCKHBW Instruction	440

PUNPCKHDQ Instruction	442
PUNPCKHWD Instruction	444
PUNPCKLBW Instruction	446
PUNPCKLDQ Instruction	448
PUNPCKLWD Instruction	450
PUSHFD Instruction	507
PWT Instruction	173
PXOR Instruction	452

## R

Ranges, Operating	303
Ratings, Absolute	304
Read and Write	
basic I/O	199
misaligned I/O	200
Reads, Burst Reads and Pipelined Burst	195
Reciprocal Square Root, Square Root and	498
Register	
boundary scan	273
bypass (BR)	277
control	34
data types, floating-point	30, 85, 89, 506
debug	36, 283
EAX	513, 515–516, 518
execution units	460
floating-point	27
general-purpose	23
SYSCALL/SYSRET target address (STAR)	41
Register Set	
3D	84
MMX	350
Register X	
and Y Execution	461
and Y Functional Units	20
and Y Pipelines	20
Execution Pipeline	91, 458
Unit	89–90, 466
Register Y	
Execution Pipeline	91, 458
Unit	89–90, 467
Registers	8, 23, 229, 256
3D	23, 31, 84, 88, 91, 93
descriptors and gates	48
device identification (DIR)	276
DR3–DR0	286
DR5–DR4	286
DR6	287
DR7	287
EFLAGS	33
extended feature enable register (EFER)	41
IR	273
MCAR	39
memory management	42
MMX	23, 31, 84, 350
segment	26
STAR	41
TAP	273
TR12	40
WHCR	42
Regulator, Voltage	335
Replacement, Cache-Line	240, 248
Requirements, Pin Connection	301
Reserved	175



RESET	174, 228, 292
and test signal timing	324
signals sampled during	227
state of processor after	229
Resource Constraints	466
Return Address Stack	22
Revision Identifier, SMM	262
RISC86 Microarchitecture	6
RSM Instruction	263, 266–267
RSVD	175

## S

Sample Code	514
SAMPLE/PRELOAD Instruction	278
Sampled	139
Scheduler	
centralized	16
instruction control unit	8
SCYC	175
Sector, Write to a	241
Segment	
descriptor	26, 48–50
registers	26
task state	44
usage	26
Selectable Drive Strength	310
Shift-DR state	280
Shift-IR state	280
Shutdown Cycle	222
Signal	
descriptions	137
switching characteristics	313
terminology	139
timing, RESET and test	324
Signals	
A[20:3]	310
A[31:3]	141
A20M#	140, 258
ADS#	142, 310
ADSC#	142
AHOLD	143, 292
AP	144
APCHK#	145
BE[7:0]#	146
BF[2:0]	147, 296
BOFF#	148, 213
BRDY#	149
BRDYC#	150, 310
BREQ	151
CACHE#	152, 237
cache-related	238
CLK	152
D/C#	153
D[63:0]	154
DP[7:0]	155
EADS#	156
EWBE#	157, 292
FERR#	158, 256
FLUSH#	159, 227, 248, 270, 292
HIT#	160
HITM#	160, 310
HLDA	161
HOLD	162
IGNNE#	163, 256

INIT	164, 292
INTR	165, 292
INV	165
KEN#	166
LOCK#	167
M/IO#	168
NA#	169
NMI	170, 292
output	229
PCD	171
PCHK#	172
PWT	173
RESET	174, 292
RSVD	175
sampled during RESET	227
SCYC	175
SMI#	176, 257, 292
SMIACK#	177, 257
STPCLK#	178, 293
TAP	272
TCK	179
TDL	179
TDO	179
TMS	180
TRST#	180
VCC2DET	181
VCC2H/L#	181
W/R#	182, 310
WB/WT#	183
SIMD	9, 82, 88, 90, 348–349, 457
Single Instruction Multiple Data	
(SIMD)	9, 82, 348–349, 457
Single-Transfer Memory Read and Write	192
Six-Stage Pipeline	459
SMI#	176, 257, 292
SMIACK#	177, 257
SMM	257
base address	263
default register values	258
halt restart slot	264
I/O trap DWORD	265
I/O trap restart slot	266
operating mode	258
revision identifier	262
state-save area	260
Snoop	177, 183, 197, 248–250
Snooping	
cache	250
internal	247
Socket 7	xxv, 1
Software Environment	23
Special	
bus cycle	149, 178, 220–223, 264, 293
cycle	157, 159, 178, 186, 198
	220, 222–223, 238, 292–293
Specifications	
package	343
package thermal	331
Split Cycle	175
Square Root	95
and reciprocal square root	498
optimized 15-bit precision	499
optimized 24-bit precision	499
square root and reciprocal	498
Stack, Return Address	22

Standard Functions	506–507, 515–516
State Machine Diagram, Bus	189
State of Processor	
after INIT	232
after RESET	229
States, Cache	246
State-Save Area, SMM	260
Stop	
clock	178
clock state	223, 295–296
grant inquire state	291–293, 295
grant state	223, 293, 295
Store Unit	463, 467, 470, 490, 493
STPCLK#	178, 293
Super7	xxv, 1, 3–4
platform initiative	3
Switching Characteristics	314
100-MHz bus operation	314
66-MHz bus operation	315
input setup and hold timings for 100-MHz bus	318
input setup and hold timings for 66-MHz bus	322
output delay timings for 100-MHz bus	316
output delay timings for 66-MHz bus	320
signal	313
valid delay, float, setup, and hold timings	316
SYSCALL	39, 42, 69, 231, 512, 519
SYSCALL/SYSRET Target Address Register (STAR)	41–42
SYSRET	69, 512, 519
System	
design, airflow management in a	336
management interrupt (SMI#)	176
management interrupt active (SMIACT#)	177
management mode (SMM)	257

## T

Table, Branch History	21
TAP	271
TAP Controller States	
capture-DR	280
capture-IR	280
shift-DR	280
shift-IR	280
state machine	278
test-logic-reset	280
update-DR	280
update-IR	280
TAP Instructions	277
BYPASS	278
EXTTEST	277
HIGHZ	278
IDCODE	278
SAMPLE/PRELOAD	278
TAP Registers	273
instruction register (IR)	273
TAP Signals	272
Target Cache, Branch	21
Task	
state segment	44
switching	82, 84, 93, 356
TCK	179
TDI	179
TDO	179

Temperature	303, 331–332, 334
case	334
Terminology, Signals	139
Test	
access port, boundary-scan	271
and debug	269
clock	179
data input	179
data output	179
logic-reset state	280
mode select	180
mode, tri-state	270
register 12 (TR12)	40
reset	180
Testing for	
extended functions	513
the CPUID instruction	506
Thermal	307, 332–336
design	331
heat dissipation path	334
layout and airflow consideration	335
measuring case temperature	334
package specifications	331
Throughput, Latencies and	465
Time Stamp Counter	40, 511–512, 517, 519
Timing Diagrams	187–226
test signal	330
TLB	7, 171, 234, 239, 270, 475, 514, 520–521
TMS	180
TR12	39–40, 231, 236–237, 243, 282
Transition from Protected Mode to Real Mode, INIT-Initiated	225
Translation Lookaside Buffer (TLB)	45, 233
Trap Dword, I/O	265
Tri-State Test Mode	270
TRST#	180
TSC	39–40, 231, 292–293
TSS	44, 50–51, 261, 287

## U

Unit	
3D	482, 500
branch condition	464
fetch	12
floating-point	482, 500, 502, 511–512
load	91, 462, 467, 469–471, 489, 491–492
scheduler/instruction control	8
store	463, 467, 470, 490, 493
see also Execution Unit	
Units	
register execution	460
register X and Y	20

## V

Values Returned by the CPUID Instruction	523
VCC2DET	181
VCC2H/L#	181
Voltage	181, 188, 299, 303, 305, 310, 314
ranges	310
regulator	335

## W

W/R#	182, 310
WAE15M	242
WAE1M	242
WB/WT#	183
WBINVD Instruction	248, 251
WCDE	42, 242, 244
WHCR	39, 42, 231, 242, 244
Write	
handling control register (WHCR)	42
to a cacheable page	241
to a sector	241
Write Allocate	236, 240–242, 244, 246
enable	42, 242
enable limit	42, 242
limit	242
logic mechanisms and conditions	243–244
Write/Read	182

Writeback	152, 154–155, 166, 173, 177
	183, 186, 197–198, 220, 233, 239
	246, 249, 251, 297
burst	197
cache	6, 10
cycles	140, 142–143, 157, 160, 183, 198
	205, 209–210, 212, 214, 216
	236–237, 282, 295
or writethrough	183
Writethrough vs. Writeback Coherency States	251

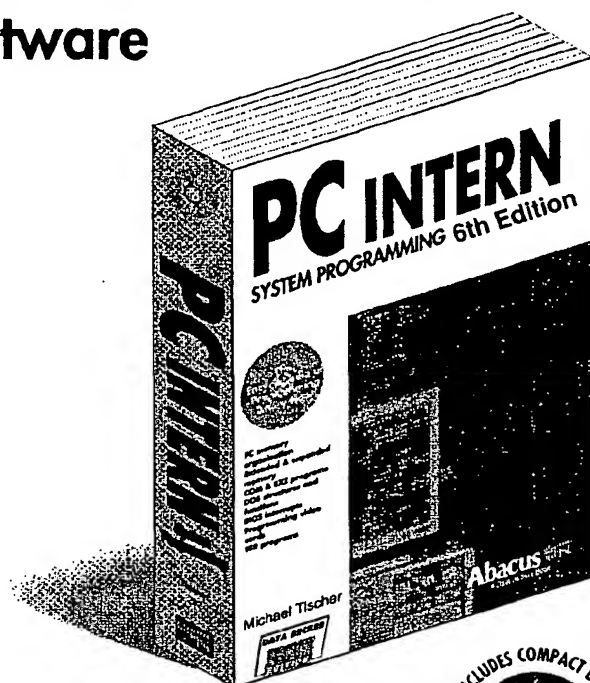
## X

x86	
coding optimizations, general AMD-K6 processor	474
coding optimizations, integer	478
optimization techniques, general	472

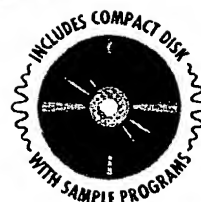


# PC catalog

**Order Toll Free 1-800-451-4319**  
**Books and Software**



**Abacus**   
[www.abacuspublisher.com](http://www.abacuspublisher.com)



**To order direct call Toll Free 1-800-451-4319**

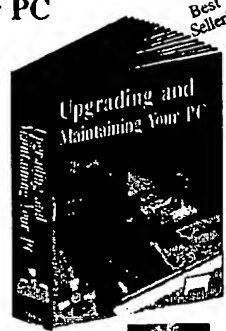
In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 6% sales tax.

Productivity Series books are for users who want to become more productive with their PC.

## Upgrading and Maintaining Your PC New Sixth Edition!

Buying a personal computer is a major investment. Today's fast-changing technology and innovations, such as Windows NT, ISDN cards and super fast components, require that you upgrade to keep your system current. New hardware and software developments require more speed, more memory and larger capacities. This book is for the millions of PC owners who want to retain their sizable investment in their PC system by upgrading.

With current info on the newest technology, *Upgrading & Maintaining Your PC* starts by helping readers make informed purchasing decisions. Whether it's a larger hard drive, more memory or a new CD-ROM drive, you'll be able to buy components with confidence.



CD-ROM  
INCLUDED!

### Inside this new 6th Edition:

- Over 200 Photos and Illustrations
- Upgrader's guide to shopping for PC motherboards, operating systems, I/O cards, processors and more!
- Windows NT Workstation 4.0, Windows 95 and OS/2 Warp 4.0
- Processors (Intel, Cyrix, AMD and more), Internal/External cache
- The latest video and sound cards and installation tips
- **SPECIAL WINDOWS 95 SECTION!**

### On the CD-ROM-

• **Wintune**—Windows Magazine's system tune-up program • **SYSINFO**—system "quick glance" program • **Cyrix Test**—Cyrix upgrade processor test • **P90 TEST**—the famous Intel Pentium "math" test • **WinSleuth**—Windows diagnostic utility • **And Much More!**

Publisher: Abacus  
Order Item #S325  
ISBN: 1-55755-329-7

Suggested Retail Price  
\$44.95 US/\$59.95 CAN  
CD-ROM Software Included

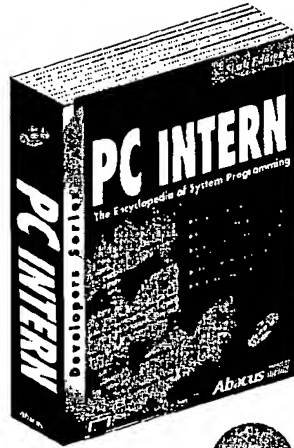
**To order direct call Toll Free 1-800-451-4319**

In US & Canada Add \$5.00 Shipping and Handling  
Foreign Orders Add \$13.00 per item. Michigan residents add 6% sales tax.

Developers Series books are for professional software developers who require in-depth technical information and programming techniques.

**PC Intern—6th Edition**  
**The Encyclopedia of System Programming**

Now in its 6th Edition, more than 500,000 programmers worldwide rely on the authoritative and eminently understandable information in this one-of-a-kind volume. You'll find hundreds of practical, working examples written in assembly language, C++, Pascal and Visual Basic—all professional programming techniques which you can use in your own programs. PC INTERN is a literal encyclopedia for the PC programmer. PC INTERN clearly describes the aspects of programming under all versions of DOS as well as interfacing with Windows.



Some of the topics include:

- Memory organization on the PC
- Writing resident TSR programs
- Programming graphic and video cards
- Using extended and expanded memory
- Handling interrupts in different languages
- Networking programming NetBIOS and IPX/SPX
- Win95 virtual memory and common controls
- IRQs—programming interrupt controllers
- Understanding DOS structures and function
- Using protected mode, DOS extenders and DPMI/VCPI multiplexer
- Programming techniques for CD-ROM drives
- Programming Sound Blaster and compatibles

Includes CD-ROM  
with Sample Programs

The companion CD-ROM transforms PC INTERN from a book into an interactive reference. You can search, navigate and view the entire text of the book and have instant access to information using hypertext links. Also included on the CD-ROM are hundreds of pages of additional programming tables and hard-to-find material.

**Author: Michael Tischer and Bruno Jennrich**

**Order Item: #B304**

**ISBN: 1-55755-304-1**

**SRP: \$69.95 US/\$99.95 CAN**

**with companion CD-ROM**

**To order direct call Toll Free 1-800-451-4319**

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 6% sales tax.

**Productivity Series**  
become more productive with your PC

## **Nets and Intranets With Win95 Getting Connected**

Windows 95 has a surprisingly rich set of networking capabilities. Built-in networking delivers an affordable and easy way to connect with others and benefit by sharing resources—files, printers, and peripherals. Network sharing saves you and your organization time and money and adds convenience.

Another great benefit of Windows 95 Networking is its ability to let you run an Intranet. This book and companion CD-ROM has all the pieces that you'll need to set up your own internal World Wide Web server (Intranet) without the expense of using an outside Internet Service Provider.

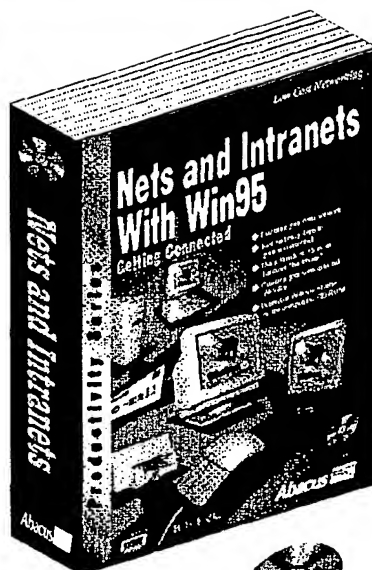
- A practical hands-on guide for setting up a small network or Intranet using Win95 or Windows for Workgroups 3.11.
- Take advantage of Windows 95's built in options so you can immediately use its networking features—
  - Shared printers
  - Easy-to-use groupware
  - E-mail and faxes
  - Additional hard drive capacity
  - Centralized backups
  - TCP/IP
- Step-by-step guide to getting and staying connected whether you're in a small office, part of a workgroup, or connecting from home.
- Perfect for the company wanting to get connected and share information with employees inexpensively

**Author: H.D. Radke**

**Item #: B311**

**ISBN: 1-55755-31-4**

**SRP: \$39.95 US/\$4.95 CAN**  
**with CD-ROM**



**CD-ROM  
Included**

**Order Direct Toll Free 1-800-451-4319**

In US and Canada add \$5.00 shipping and handling. Foreign orders add \$13.00 per item.  
Michigan residents add 6% sales tax.

Productivity Series books are for users who want  
to become more productive with their PC.

## McAfee Anti-Virus for Beginners

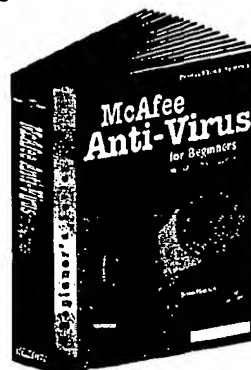
In 1986 there was one virus; today there are 7400. *McAfee Anti-Virus for Beginners* offers an introduction to protecting and securing your system from viruses.

The first step to creating a safe system is understanding what viruses are, how they replicate, how they're contracted, how they hide from detection, what 'triggers' are, and what the 'payload' is—what happens when a virus is activated. You'll learn how to recognize the symptoms of a virus, then learn to use two of McAfee's most popular products: VirusScan and WebScan.

This book will show you how to install and configure fully-functional evaluation versions of both VirusScan and WebScan from the CD-ROM. You'll learn how to use VirusScan to detect and remove viruses that may already be present, and how to protect your system in the future.

Topics discussed include:

- What viruses are
- How anti-virus software works
- Installing & configuring anti-virus software
- Detecting viruses and cleaning your system
- Internet virus concerns
- The importance of regular anti-virus software updates



CD-ROM  
INCLUDED!

### On the CD-ROM-

The companion CD-ROM contains fully functional evaluation versions of McAfee's most popular anti-virus programs: VirusScan and WebScan. Use these programs to safeguard your PC on and off the Internet.

Author: Brian Howard  
Order Item #B318  
ISBN: 1-55755-318-1

Suggested Retail Price  
\$19.95 US/\$26.95 CAN  
CD-ROM Included

**To order direct call Toll Free 1-800-451-4319**

In US & Canada Add \$5.00 Shipping and Handling  
Foreign Orders Add \$13.00 per item. Michigan residents add 6% sales tax.



# ***About This Book and CD-ROM***

## **The Book**

At the time of publication, AMD had not made final naming decisions for the processor and the 3D technology. The names used in this book are the AMD code names for the processor and the 3D technology.

Refer to Appendix B, “Code Optimization” on page 455 for details regarding the examples shown in the AMD-K6 3D simulator, especially the tables beginning with Table 87 on page 468.

Refer to the AMD web site at [www.amd.com](http://www.amd.com) for updates to material related to this book, including new scripts and updates for the AMD-K6 3D simulator.

## **CD-ROM Contents**

The CD-ROM included with this book contains the following:

- AMD-K6 3D processor simulator
- All AMD processor technical documentation in Adobe Acrobat PDF format
- Adobe Acrobat Reader for most platforms

## **Minimum System Requirements**

The following minimum system is required in order to run the AMD-K6 3D processor simulator:

- A 133-MHz AMD-K6 processor
- 16 Mbytes of memory
- Windows® 95 or Windows NT™ 4.0 operating system
- 256-color SVGA graphics mode video
- Video resolution of 800 by 600

We recommend the following system for maximum enjoyment when using the simulator:

- A 166-MHz AMD-K6 processor or better
- 32 Mbytes of memory
- Windows 95 or Windows NT 4.0 operating system
- 65,536-color graphics mode or better
- Video resolution of 1024 by 768
- A sound card with speakers

## **Installation of the CD-ROM**

A setup program is provided for your convenience. Run `install.exe` from the root directory of the CD-ROM, and the installation program will step you through installing the simulator and the Acrobat reader, if you need it.

## Installing the CD-ROM

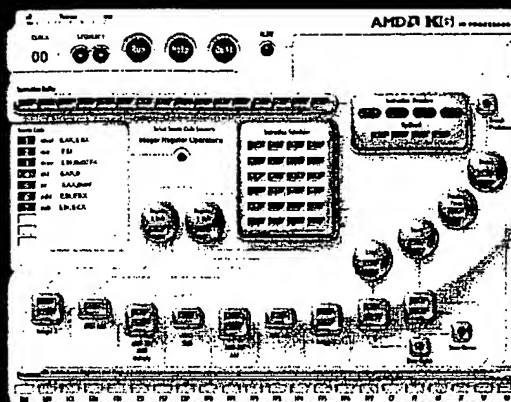
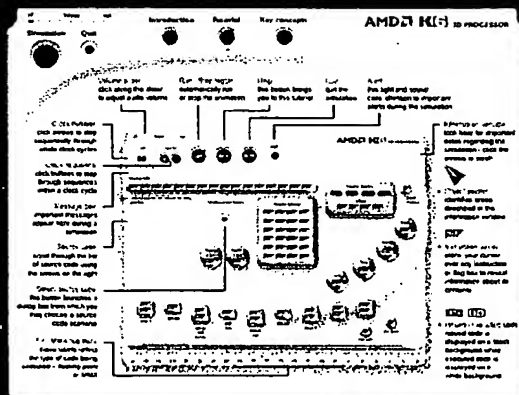
A setup program is provided for your convenience. Run `install.exe` from the root directory of the CD-ROM, and the installation program will step you through installing the simulator and the Acrobat reader if you need it.

### The AMD-K6 3D Processor Simulator

The AMD-K6 3D processor simulator is a multimedia application that enables you to see a conceptual view of x86 code as it flows through the internal functional units of the processor. The simulator includes a tutorial/help system. You may select from several scenarios that animate code flowing through the processor.

The simulator also has a script authoring mode that allows you to write your own scripts. Using the scripting language is explained in detail in the Scripting Users Guide. The users guide is accessed by opening the file named `aindk6script.pdf` found on the CD-ROM in the `\AMDK6SIM` directory.

Refer to Appendix B, "Code Optimization," for details regarding the examples shown in the AMD-K6 3D simulator; especially the tables beginning with Table 87 on page 468.



## AMD Processor Technical Documentation

The CD-ROM also includes a complete set of all technical documentation from the AMD Computation Products Group. You can access this set of documents via a custom hypertext-linked menu system that makes use of the features of Adobe Acrobat Reader. To make the documentation even more useful, the documents on the entire CD-ROM have been indexed and are completely keyword searchable. This feature allows you to find every occurrence of your search term in all documents on the CD-ROM.

The AMD technical documents are located in the `\DOCS` directory on the CD-ROM. Double-click on the `open_me.pdf` file to begin viewing the technical documents.

### Adobe Acrobat Reader

Special versions of the Adobe Acrobat Reader are included on the CD-ROM for most platforms. These special versions of the reader include the search plug-in so you may utilize the global search functions.

# This book describes the new AMD-K6-2 Processor! Revolutionary Multimedia Performance

The AMD-K6 3D Processor is written for PC users interested in the latest advance in the computer industry. The 3D graphic capability of the AMD-K6 3D is truly revolutionary, delivering on the promises made but never realized by MMX™—dramatic and real improvements in multimedia performance at the desktop.

With AMD-K6 3D technology, new, more powerful hardware and software applications enable a more entertaining and productive PC platform. Improvements include faster frame rates on high-resolution scenes, superior modeling of real-world environments and physics, sharper and more detailed 3D imaging, smoother video playback and near-theater-quality audio.

The AMD-K6 3D Processor describes the operation of the AMD-K6 3D CPU. In addition to the 3D graphic capability, you'll learn the internal architecture of the AMD-K6-2 processor with hundreds of illustrations, charts and tables.

The AMD-K6 3D Processor includes:

- The AMD-K6 3D instruction set definitions
- Examples of how the AMD-K6 3D signals interact with external devices
- Definitions of the complete MMX instruction set
- Examples showing the internal operation of the processor



The CD-ROM features an interactive simulation showing the operation of the AMD-K6 3D processor as it executes instructions. See for yourself how AMD is making your PC SCREAM!!!



You can count on us  
**Abacus**

5370 52nd Street SE Grand Rapids, MI 49512  
www.abacuspub.com

## Revolutionary Multimedia Performance

The AMD-K6 3D Processor:

Learn how it boosts your PC video and multimedia performance by 50%.

Learn powerful solutions for creating a more entertaining and productive PC platform.

Break through existing bottlenecks with multimedia and floating-point-intensive applications.

"AMD listened to their customers, and they have implemented a real improvement to the x86 instruction set. You will be able to see this improvement in the performance of the AMD-K6 3D processor. The improvement is not trivial."

John C. Dvorak

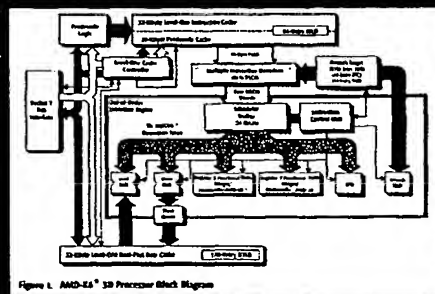


Figure 1. AMD-K6 3D Processor Block Diagram

ISBN 1-55755-345-9



9 781557 553454

\$34.95 U.S.  
\$46.95 CAN

Computer Book Category

IBM/PC: General Computing  
Level: Intermediate-Advanced